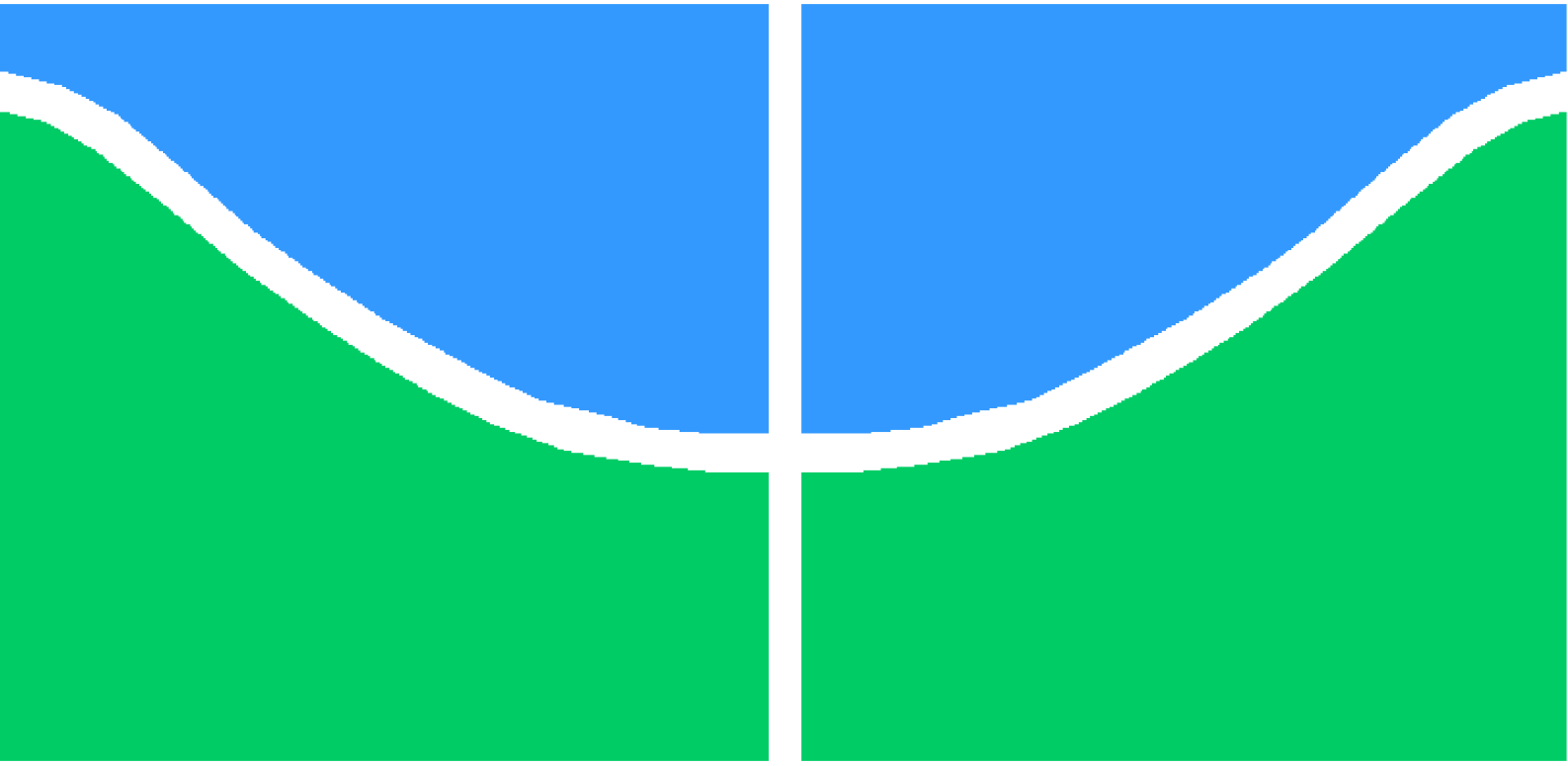

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

Equalização de Frequência em Cifradores de Fluxo: Uma proposta de algoritmo

Autor: Gabriela Matias Navarro
Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Brasília, DF
2014

Gabriela Matias Navarro

Equalização de Frequência em Cifradores de Fluxo: Uma proposta de algoritmo

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software .

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Brasília, DF

2014

Gabriela Matias Navarro

Equalização de Frequência em Cifradores de Fluxo: Uma proposta de algoritmo/ Gabriela Matias Navarro. – Brasília, DF, 2014-

121 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Luiz Augusto Fontes Laranjeira

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2014.

1. balanceamento. 2. cifra de fluxo. I. Prof. Dr. Luiz Augusto Fontes Laranjeira. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Equalização de Frequência em Cifradores de Fluxo: Uma proposta de algoritmo

CDU 02:141:005.6

Gabriela Matias Navarro

Equalização de Frequência em Cifradores de Fluxo: Uma proposta de algoritmo

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software .

Trabalho aprovado. Brasília, DF, 25 de novembro de 2014:

Prof. Dr. Luiz Augusto Fontes Laranjeira
Orientador

Prof. Dr. Fabricio Ataides Braz
Convidado 1

Prof. Dr. Edson Alves da Costa Júnior
Convidado 2

Brasília, DF
2014

Dedico este trabalho ao meu avô Balbino,
que por muito pouco não conseguiu comemorar
o fim desta jornada comigo.

Agradecimentos

Agradeço ao professor doutor Luiz Laranjeira por ter aceito ser meu orientador e pelas suas ideias, que tornaram possível a finalização deste trabalho.

Agradeço aos meus pais, por todo o carinho e apoio que me deram durante minha vida. Ao meu pai, João, pelos conselhos, incentivo e ajuda, não somente durante a minha graduação, mas em todos os momentos de minha vida. À minha mãe, Rosene, pela paciência e amor durante os momentos críticos desta jornada.

Agradeço aos meus irmãos, por sempre estarem do meu lado. Ao meu irmão Lucas, que apesar de nos desentendermos muito, sei que nos amamos e nos apoiamos. Ao meu irmão Victor, que sempre foi muito paciente e amoroso.

Agradeço aos meus avôs, que sempre apoiaram minhas decisões e sempre me deram palavras de conforto e incentivo.

Agradeço aos professores da FGA, que transmitem seus conhecimentos aos alunos na esperança de que nos tornaremos profissionais melhores.

E por fim, agradeço aos amigos que fiz durante a minha graduação, em especial o Matheus Tristão, Charles de Oliveira e aos amigos do laboratório LAPPIS.

Do what you can,
with what you have,
where you are.
Theodore Roosevelt

Resumo

A criptografia tem duas formas principais para cifrar um texto utilizando chave simétrica e chave assimétrica. Os algoritmos que utilizam chave simétrica são divididos em algoritmos de cifra de bloco e cifra de fluxo. Uma falha muito explorada por atacantes que desejam quebrar um texto cifrado é da análise de frequência de ocorrência dos caracteres do mesmo, pois a frequência média de ocorrência é conhecida para cada língua e muitos dos algoritmos não se preocupam com o balanceamento dessa frequência enquanto cifrando o texto. Este trabalho de conclusão de curso apresenta uma proposta de algoritmo que irá realizar o balanceamento completo da frequência de caracteres, aumentando sua segurança contra atacantes e curiosos.

Palavras-chaves: criptografia. cifra de fluxo. balanceamento.

Abstract

The cryptography has two ways to encrypt a text using symmetric key and asymmetric key. The algorithm using symmetric key algorithms are divided into block cipher and stream cipher. A failure much exploited by attackers who wish to break a cipher text is the analysis of frequency of occurrence of the same characters, because the medium frequency of occurrence is known for each language and many of the algorithms do not worry about balancing this frequency while encrypting the text. This course conclusion work proposes a algorithm that will perform the complete balancing of the frequency of characters, increasing their security against attackers and curious.

Key-words: crytography. stream cipher. balancing.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

DES	Data Encryption Standard.
IBM	International Business Machines.
NIST	National Institute of Standards and Technology
AES	Advanced Encryption Standard
ECB	Electronic codebook
CBC	Cipher-block chaining
CFB	Cipher feedback
OFB	Output Feedback
CTR	Counter
SSL	Secure Sockets Layer
GSM	Global System for Mobile
LFSR	Linear feedback shift register
TLS	Transport Layer Security
WEP	Wired Equivalent Privacy
RSA	Algoritmo criptográfico de chave assimétrica
MIT	Massachusetts Institute of Technology
TRNG	True random number generator
PRNG	Pseudorandom number generator
RC4	Rivest Cipher 4

Lista de símbolos

$\oplus$	Símbolo lógico ou-exclusivo
ϕ	Função Totiente de Euler

Sumário

1 Introdução

Desde tempos remotos, segredos sempre existiram e métodos para deixá-los indecifráveis foram construídos. Enquanto algumas pessoas estavam escondendo informações, outras curiosas estavam tentando descobri-los (MAXIMOV, 2006).

Na constante luta entre manter e quebrar segredos, métodos foram desenvolvidos para se esconder informações e falhas nesses métodos foram exploradas para se obter as informações.

Um dos métodos para obter informações sem a autorização necessária é a análise de frequência da ocorrência de caracteres dessas informações escondidas. Há estudos que indicam qual a frequência média de utilização de cada letra em um texto. Analisando um texto cifrado e calculando a frequência de ocorrência dos caracteres do mesmo, pode ser possível, através da substituição dos mesmos pelos caracteres com frequência similar da língua, quebrar a cifragem utilizada e ter acesso à mensagem original.

1.1 Problema

Como há a possibilidade de se obter as informações simplesmente analisando a frequência de ocorrência de caracteres, isso torna a comunicação insegura. Como consequência, o principal objetivo de se utilizar o algoritmo criptográfico, que é proteger informações importantes, torna-se obsoleto.

1.2 Objetivos do Trabalho

1.2.1 Objetivo Geral

Esse trabalho visa apresentar um algoritmo de criptografia que dificulte a análise de frequência de um texto permitindo assim a comunicação privada e segura.

1.2.2 Objetivos Específicos

- Realizar a proposta de algoritmo.
- Implementar o algoritmo proposto.
- Realizar experimentos, voltados para a análise de desempenho, afim de compará-los com resultados do algoritmo RC4.

1.3 Organização do Trabalho

Este trabalho tem a seguinte organização:

No Capítulo 2 são apresentados os principais tipos de algoritmos criptográficos, incluindo suas descrições e exemplos, assim como são descritos métodos de criptoanálise.

No Capítulo 3 é descrito o que é um gerador de números pseudo-aleatórios, alguns exemplos e suas implementações e é explicada a importância desses geradores para a criptografia.

No Capítulo 4 é explicada a proposta do algoritmo com suas soluções, seu funcionamento e as principais vantagens em relação a outros algoritmos.

No Capítulo 5 é explicado como foi feito a implementação com a arquitetura que foi utilizada e os passos para utilizar o algoritmo.

No Capítulo 6 é explicado os experimentos realizados e como foi feita a coleta de dados desses experimentos.

No Capítulo 7 é apresentada a conclusão deste trabalho, a descrição dos trabalhos futuros e suas importâncias.

2 Criptografia

A criptografia é a ciência dos códigos secretos, permitindo a confidencialidade de uma comunicação que utiliza um meio inseguro (VAUDENAY, 2006). A necessidade de manter informações em sigilo impulsionou a evolução dos estudos da criptografia. O exato início da criptografia é incerto, mas na Renascença, assim como outros em muitos outros campos, o estudo da criptografia começou a ser aprofundado e suas técnicas armazenadas e ensinadas (DAVIES, 1997).

Uma das primeiras formas de criptografia utilizada foi a substituição de caracteres. A substituição primeiramente era feita de forma fixa, ou seja, era usada uma única tabela em que as letras do alfabeto eram trocadas por outras letras. Leon Battista Alberti introduziu uma maneira diferente de fazer a substituição, criando o algoritmo *polyalphabetic substitution*. O princípio desse algoritmo era a criação de várias tabelas de substituição e a utilização de uma chave para definir a ordem que cada tabela seria usada para cifrar cada letra do texto em claro. O conjunto de tabelas de substituição é chamado de *Vigenere tableau*.

Durante a II Guerra Mundial, informações relacionadas às estratégias de guerra deveriam ser mantidas em sigilo. Para isso, duas máquinas foram criadas e merecem destaque: a *Enigma* e a *Lorenz SZ40*. A máquina *Enigma* foi baseada nas capacidades criptográficas de uma série de motores conectados por fio e *plugboard* (WILCOX, 2006). O *plugboard* fazia a substituição de uma letra por outra letra.

Figura 1 – Máquina Enigma[1]

A máquina *Lorenz* foi um impressor anexado que automaticamente cifrava o texto em claro digitado pelo operador e enviava o texto cifrado por um fio de comunicação. No

[1] Disponível e adaptado de: <http://www.bbc.co.uk/history/topics/enigma>

outro lado da comunicação, tinha uma máquina *Lorenz* idêntica que fazia a decifração automaticamente e imprimia o texto em claro no papel (COLLINS, 2006).

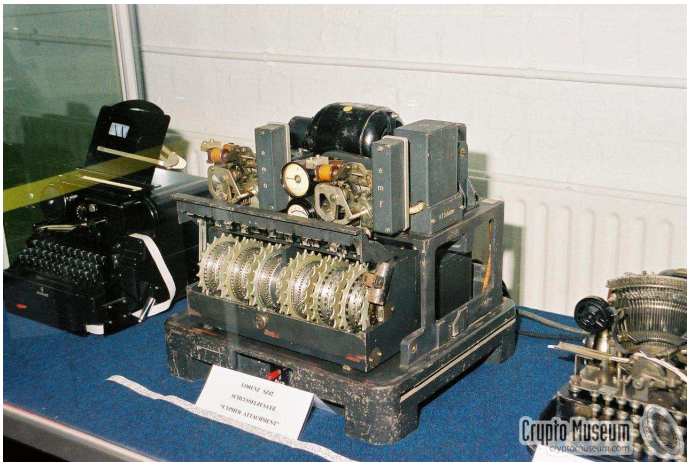

Figura 2 – Máquina SZ40[2]

Após a guerra, a publicação do algoritmo *DES* serviu de incentivo para cientistas do mundo todo aprofundarem suas pesquisas em novos algoritmos. Nos dias de hoje existem duas linhas de estudo sobre a criptografia, a criptografia com chave simétrica e a com chave assimétrica.

2.1 Criptografia de Chave Simétrica

A criptografia simétrica utiliza a mesma chave[3] para cifrar[4] e decifrar[5] uma mensagem. É objeto de estudo para muitos cientistas e existem muitos algoritmos que ainda são utilizados pelo mundo. Sua vantagem é que o custo para ser utilizada não é tão elevado e sua possibilidade de uso é maior.

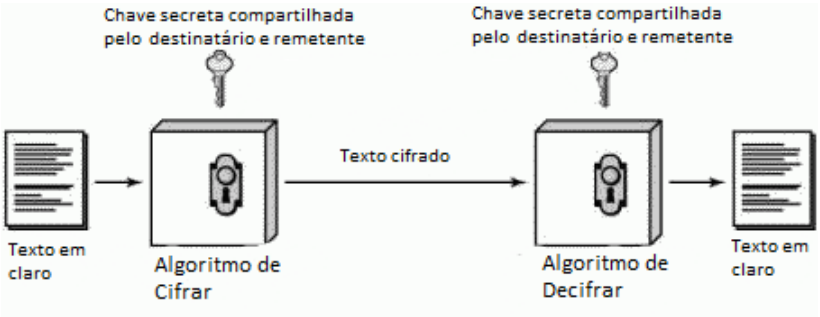

Figura 3 – Funcionamento básico da criptografia simétrica[6]

[2] Disponível e adaptado de: <http://www.cryptomuseum.com/crypto/lorenz/sz40/>

[3] Entrada para as funções cifrar e decifrar, é usada para que duas pessoas consigam se comunicar em segredo.

[4] Função que tem como objetivo transformar uma mensagem que deve estar em sigilo e transformá-la em algo ininteligível.

[5] Função que transforma uma mensagem ininteligível para a mensagem original.

Os algoritmos são divididos em: cifra de bloco e cifra de fluxo.

2.1.1 Cifra de Bloco

A cifra de bloco tem como princípio a cifra de blocos de caracteres com tamanho definido e a mensagem criptografada deve ter o mesmo tamanho da mensagem em claro. Um algoritmo muito conhecido é o *DES* que foi desenvolvido pela *IBM*. Foi a primeira cifra comercialmente desenvolvida e sua estrutura foi divulgada por completo (BIRYUKOV, 2011).

O tamanho da chave utilizada no *DES* é considerado um problema nos dias atuais. Muitos pesquisadores sabendo desse problema tentaram remediá-lo com novas variantes criptográficas. Um exemplo é o *3-DES* que utiliza duas chaves no processo, mas que, como consequência, tem seu desempenho prejudicado.

Outra medida tomada para contornar o problema de ataques ao *DES* foi a publicação de um concurso do *NIST* para a escolha de um novo padrão criptográfico. Assim surgiu o *AES*, que é um algoritmo que utiliza o sistema de permutação e substituição, tem o tamanho do bloco fixo de 128 bits e tamanho de chave variável entre 128, 192 ou 256 *bits*.

Existem operações que podem ser aplicadas nas cifras de blocos e que podem ser utilizadas nos algoritmos *DES* e *AES* para a produção de mensagens criptografadas mais seguras, são elas:

ECB a mensagem é dividida em blocos e consiste em cifrar os blocos de forma independente um do outro e com uma chave fixa. A desvantagem desse modo de cifra é que blocos de texto em claro iguais produzem blocos de texto cifrado iguais.

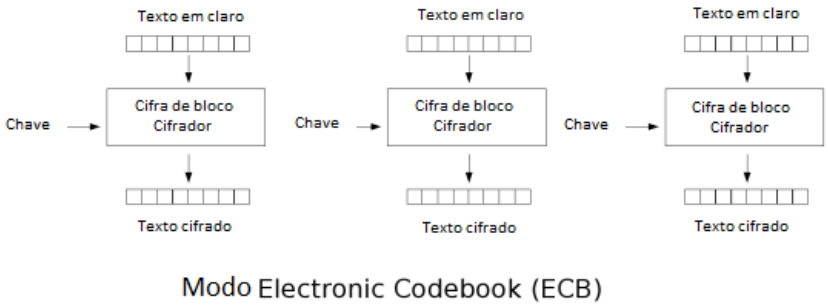

Figura 4 – Modo *Electronic Codebook*[7]

[6] Disponível e adaptado de: <http://www.codeproject.com/Articles/21076/Securing-Data-in-NET>
[7] Disponível e adaptado de: <http://cryptodox.com/Block_cipher_modes_of_operation>

CBC a mensagem é dividida em blocos e o modo de cifra utiliza da operação de ou-exclusivo entre um vetor de inicialização (para o primeiro bloco) e o texto em claro[8] e por fim um ou-exclusivo do resultado com a chave. A partir do segundo bloco, ao invés de um vetor de inicialização, é utilizado o bloco criptografado anterior.

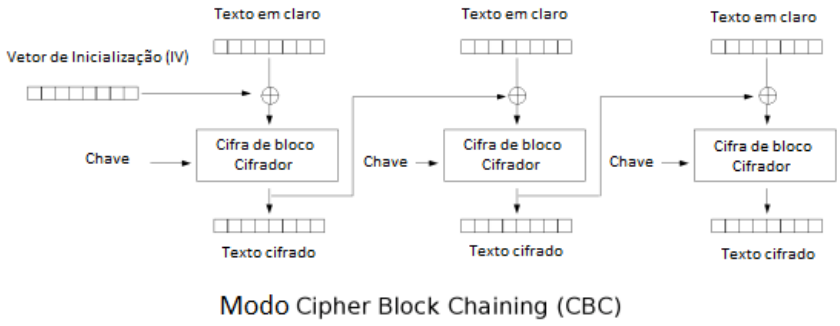

Figura 5 – Modo *Cipher Block Chaining*[9]

CFB a mensagem também é dividida em blocos e esse modo de cifra é muito similar ao *CBC*, porém é feito um ou-exclusivo da chave com o vetor de inicialização, o bloco criptografado anterior e com o resultado é feito um ou-exclusivo com a mensagem em claro.

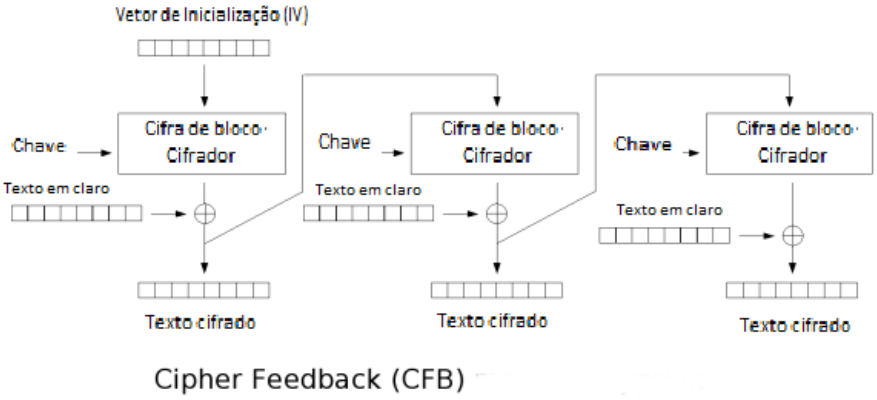

Figura 6 – Modo *Cipher Feedback* [10]

OFB a diferença em relação ao CFB é que o resultado entre o vetor de inicialização e a chave ou resultado da operação anterior e a chave, serve de entrada para o próximo bloco, ao invés do bloco criptografado.

[8] Mensagem que deve ser mantida em sigilo e que se deseja cifrar.

[9] Disponível e adaptado de:<https://www.adayinthelifeof.nl/2010/12/08/encryption-operating-modes-ecb-vs-cbc/>

[10] Disponível e adaptado de: <http://crypto.stackexchange.com/questions/2476/cipher-feedback-mode>

[11] Disponível e adaptado de: <http://commons.wikimedia.org/wiki/File:Ofb_decryption.png>

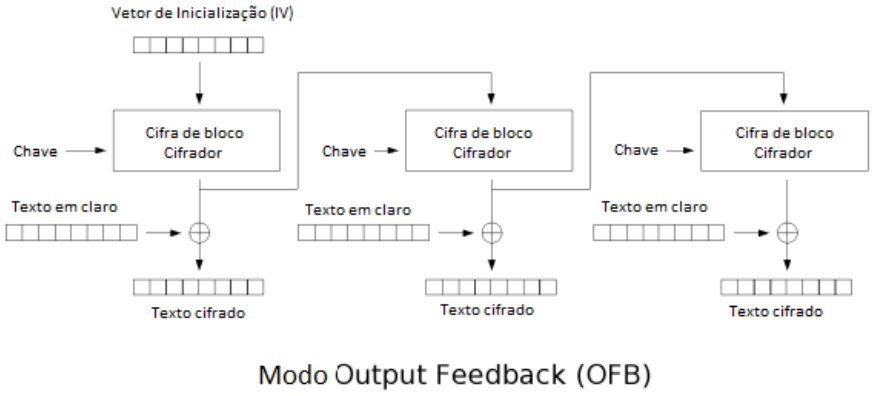

Figura 7 – Modo *Output Feedback*[11]

CTR utiliza como entrada para a função criptográfica um contador que é acrescentado de um em um.

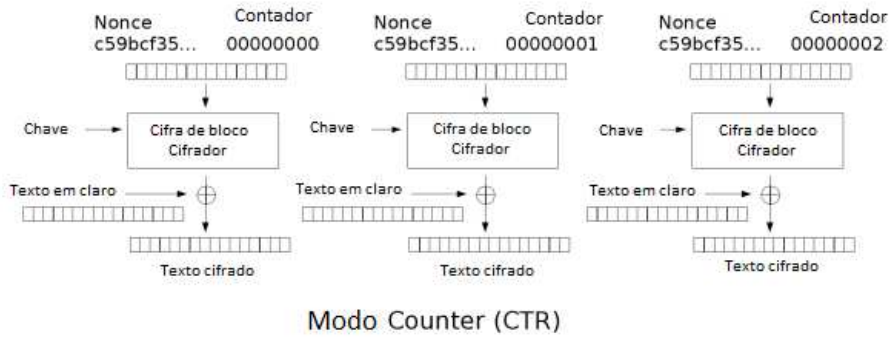

Figura 8 – Modo *Counter*[12]

2.1.1.1 Algoritmo DES

Como dito anteriormente, o DES foi o primeiro algoritmo que foi publicado e utilizado de forma comercial. Esse algoritmo utiliza o princípio da cifra de *feistel*[13] com 16 *rounds* de processamento.

Esse algoritmo utiliza chave de 56 *bits* e opera em blocos de dados de 64 *bits*. A fase de inicialização é feita uma permutação do bloco de dados iniciais. O outro passo da fase de inicialização é dividir o bloco de dados em blocos de 32 *bits*.

Depois dessa divisão, as operações são executadas no bloco que fica a direita como mostrado na Figura 9. Esse algoritmo introduz dois atributos: confusão e difusão. O atributo de confusão é feito por uma camada de substituição *S-box*. A camada de permutação

[12] Disponível e adaptado de: <http://crypto.stackexchange.com/questions/8151/counter-mode-static-iv-but-different-keys>

[13] Estrutura de criptografia simétrica que inclui dividir o bloco de dados e depois realizar operações em um deles e ao final realizar um ou-exclusivo nos dois blocos.

[14] Disponível e adaptado de: <http://www.eventid.net/show-DocId-19.htm>

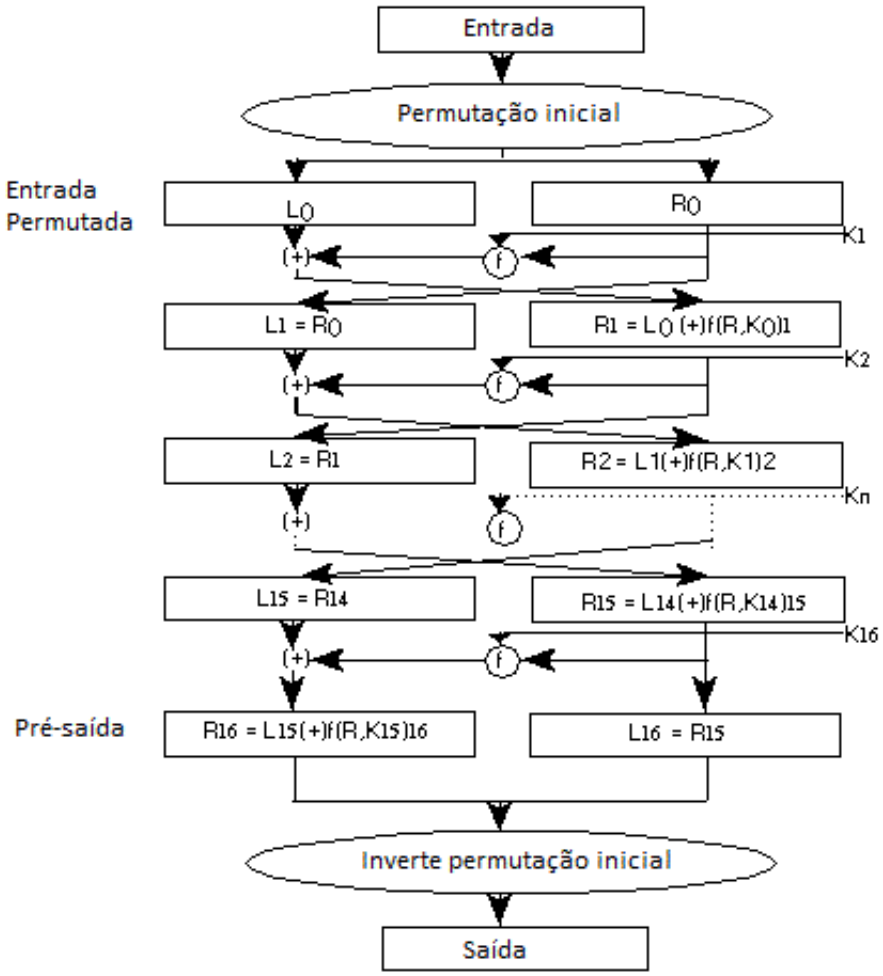

Figura 9 – Cifra DES[14]

é feita pela *P-box* e é responsável pelo atributo de difusão na cifra. O resultado dessas duas operações no bloco do lado direito é feito um ou-exclusivo com o lado esquerdo e isso resulta no lado direito do próximo *round*. Essas operações se repetem em todos os 16 *rounds*. No último passo é realizada a inversão da permutação inicial. No Pseudo-código 2.1 pode ser demonstrado os passos acima descritos.

Código 2.1 Pseudo-código DES

```
1 void desCipher(char plainBlock[64], char roundKeys[56], char
    cipherBlock[64]){
2
3   char tempBlock[64] = permutation(plainBlock, initialPermutation
      )
4   char leftBlock[32] = split(0,32,tempBlock)
5   char rightBlock[32] = split(32,64,tempBlock)
6   for(i = 0; i < 16; i++){
7     sbox = sbox(rightBlock)
```

```
8       pbox = pbox(sbox)
9       rightBlock = pbox ^ leftBlock
10    }
11
12    tempBlock = combination(leftBlock, rightBlock)
13    cipherBlock = permutation(tempBlock, finalPermutation)
14  }
```

2.1.1.2 Algoritmo AES

O algoritmo *AES* foi inventado com o intuito de dar mais segurança ao processo de criptografia e foi adotado como algoritmo padrão pelo *NIST*. Esse algoritmo, ao contrário do *DES*, não utiliza a cifra de *feistel* como base.

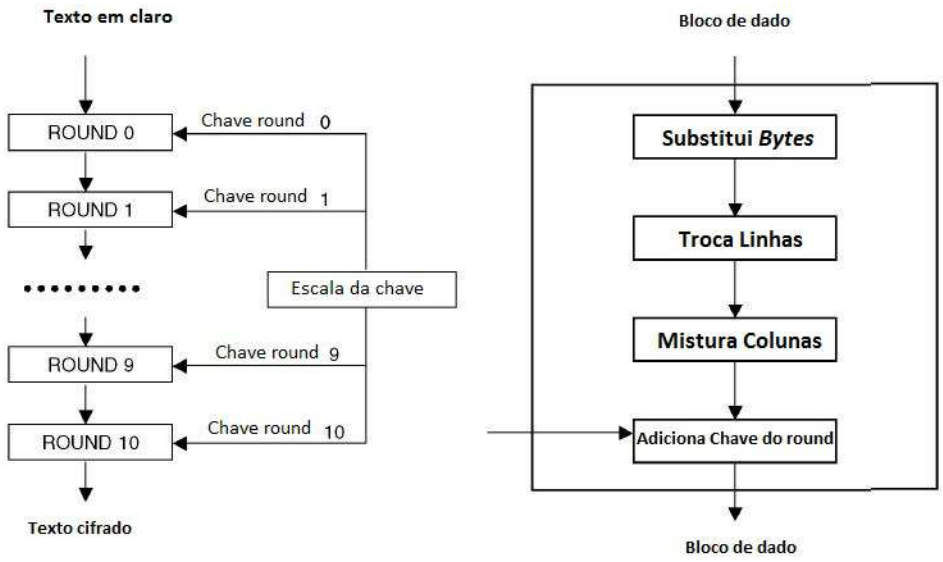

Figura 10 – Cifra AES[15]

O algoritmo *AES* utiliza chaves com tamanhos variáveis e essas dependem da quantidade de *rounds* que são utilizados. Exemplo é que para 10 *rounds* o tamanho da chave que deve ser utilizado é de 128 *bits*, para 12 *rounds* o tamanho da chave é de 192 *bits* e para 14 *rounds* o tamanho da chave correspondente é 256 *bits*. Isso é feito, pois para cada *round*, um pedaço da chave é utilizado.

O tamanho do bloco que é utilizado é de 128 *bits* e é chamado de estado. O bloco de estado é dividido em uma matriz 4×4. A forma como cada *round* é feita pode ser

[15] Disponível e adaptado de: <http://www.ecs.umass.edu/ece/koren/FaultTolerantSystems/simulator/AES-L/help.html>

vista na Figura 10. Todos os *rounds* são idênticos, com exceção do último *round*. Em cada *round* a operação de substituição com a função *S-box* é feita e essa operação introduz confusão a cifra. Outra operação que é realizada em cada *round* é a de permutação com a função de *P-box* e introduz difusão a cifra. A permutação é composta pela mistura de colunas e linhas da matriz que o bloco foi dividido. A diferença entre o último *round* e os anteriores é o passo de mistura das colunas, o documento de proposta de algoritmo explica o porque: Para que a cifra e o inverso da mesma sejam mais similares em estrutura, a camada de mistura de colunas do último *round* é diferente das camadas de permutações de *rounds* anteriores (DAEMEN; RIJMEN, 1998).

Código 2.2 Pseudo-código AES

```
1  void aesCipher(char plainState[128], char roundKeys[rounds], char
       cipherState[128]){
2
3    char matrix[4][4] = transformInMatrix(plainState)
4    for(i = 0; i < rounds; i++){
5      char tempMatrix[4][4] = sBox(matrix)
6      tempMatrix = shiftRows(tempMatrix)
7      tempMatrix = mixColumns(tempMatrix)
8      matrix = AddKeys(tempMatrix, roundKeys[i])
9    }
10   tempMatrix = shiftRows(matrix)
11   cipherBlock = AddKeys(tempMatrix, roundKeys[round])
12 }
```

2.1.2 Cifra de Fluxo

Os algoritmos que utilizam a cifra de fluxo o fazem *bit* a *bit* ou *byte* a *byte*. Uma das vantagens da cifra de fluxo, em relação a cifra de bloco, é seu desempenho superior e, por esse motivo, é muito utilizado em sistemas de redes, tais como *bluetooth*, *SSL* e outros.

Outra vantagem é a possibilidade de transformar uma cifra de bloco em cifra de fluxo. Para isto basta definir que o tamanho do bloco seja um *bit* ou um *byte*. Alguns dos algoritmos de fluxo mais populares, que hoje cobrem mais de 80% do mundo na telecomunicação e *cyber space*, são: A5/1, A5/2, E0 e RC4 (BAKHTIARI; MAAREF, 2011).

A Figura 11 representa o funcionamento da cifra de fluxo, que funciona a partir de um gerador de *keystream*[16], que é gerado a partir de uma chave. Este *keystream* é

[16] Sequência de caracteres aleatórios ou pseudo-aleatórios que são utilizados para a cifra de fluxo.

utilizado para cifrar os *bits* do texto em claro. Para decifrar, o mesmo *keystream* deve ser gerado para que possa fazer a operação ou-exclusivo com os *bits* do texto cifrado.

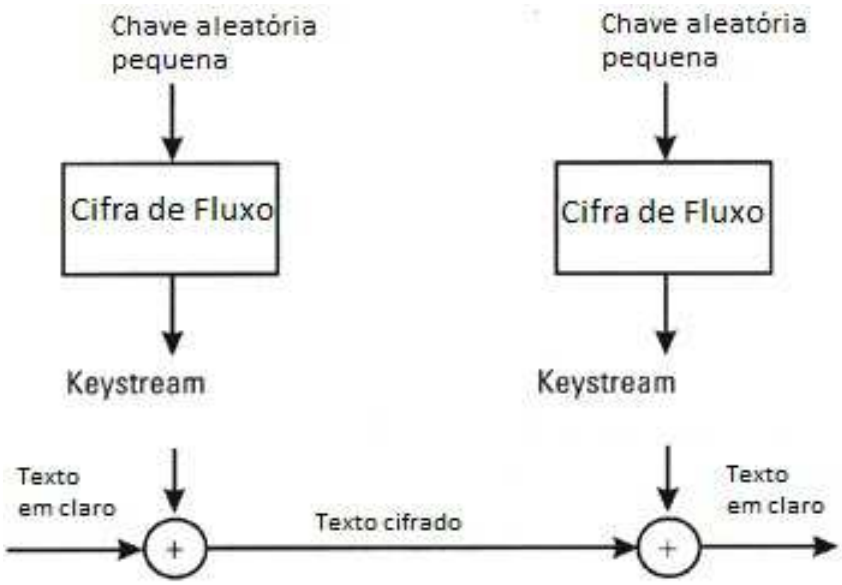

Figura 11 – Funcionamento da cifra de fluxo[17]

2.1.2.1 Algoritmo A5/1

Principal algoritmo usado no mundo para a comunicação *GSM*, foi desenvolvido em 1987 e teve seu funcionamento mantido em segredo e somente em 1999 foi revelado por engenharia reversa. Consiste de três *Linear Feedback Shift register* ou *LFSR* binários R1, R2 e R3. Um *LFSR* é um registrador que mantém valores de estado anterior e é atualizado por uma função que normalmente é um ou-exclusivo. Esses registradores são atualizados com cronômetro irregular, o que significa que a forma de atualização dos três registradores acontecem de forma independente.

Cada registrador contém *bits* importantes para a criptografia e suas posições são predeterminadas. São os *bits* do *clocking* e *tapping*, que estão definidos na tabela 1. Os registradores tem tamanhos diferentes sendo que, o R1 tem 19 *bits*, o R2 tem 22 *bits* e o R3 tem 23 *bits*. O algoritmo tem como entradas a chave secreta *Kc*, que tem 64 *bits*, e um vetor de inicialização de 22 bits composto do *frame number*. O *keystream*, de 228 *bits*, é a saída do processo, sendo que os primeiros 114 *bits* são o *keystream* de *downlink* e os outros 114 *bits* são o *keystream* de *uplink*. Cada ciclo no algoritmo é composto por passos específicos em cada registrador:

A operação de ou-exclusivo é realizada entre os *tapping bits* de cada registrador. Utilizando a regra de maioridade[19] com os *clocking bits*. O registrador é acionado se o

[17] Disponível e adaptado de: <http://www.globalspec.com/reference/81191/203279/2-6-stream-ciphers>

[18] Disponível e adaptado de: <http://cryptome.info/0001/a51-bsw/a51-bsw.htm>

[19] Essa regra funciona da seguinte maneira, os três *clocking bits* de cada registrador são comparados

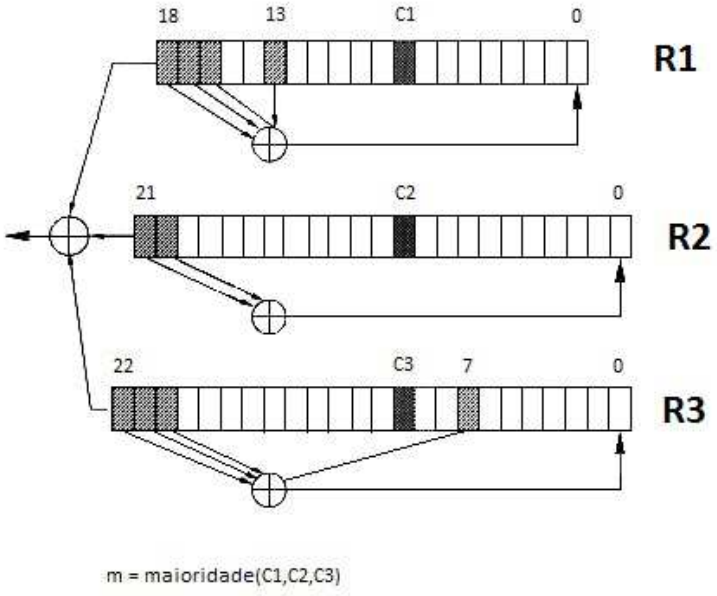

Figura 12 – Estrutura do algoritmo A5/1[18]

Registrador	*Clocking Bits*	*Tapping Bits*
R1	8	13, 16, 17, 18
R2	10	20,21
R3	10	7,20,21,22

Tabela 1 – *Bits* importantes do algoritmo A5/1

valor contido no *clocking bits* for igual ao resultado da regra de maioridade e então os *bits* do registrador são deslocados da direita para a posição mais a esquerda do registrador.

Código 2.3 Pseudo-código A5/1

```
1  void a51Generation(key key[64]){
2
3    int i;
4    for(i = 0; i < 64; i++){
5      R1[0] = R1[0] ^ key[i]
6      R2[0] = R2[0] ^ key[i]
7      R3[0] = R3[0] ^ key[i]
8    }
9
10   registers R1[19] ,R2[22], R[23]
```

e os registradores que tiverem os *bits* que teve maior ocorrência na comparação são acionados, por exemplo o R1 tem *bit* 1, o R2 tem *bit* 0 e o R3 tem *bit* 1, então o *bit* com maior ocorrência é o 1, sendo assim somente os registradores R1 e R3 são acionados naquele ciclo.

```
11
12   resultR1 = R1[13] ^ R1[16] ^ R1[17] ^ R1[18]
13   resultR2 = R2[20] ^ R2[21]
14   resultR3 = R3[7] ^ R3[20] ^ R3[21] ^ R3[22]
15
16   clockingBit = marjorityRule(R1[8], R2[10], R3[10])
17   if(R1[8] == clockingBit){
18     deslocateToTheLeft(resultR1)
19   }else if(R2[10] == clockingBit){
20     deslocateToTheLeft(resultR2)
21   }else if(R3[10] == clockingBit){
22     deslocateToTheLeft(resultR3)
23   }
24
25   output = R1[18] ^ R2[21] ^ R3[22]
26 }
```

Se o registrador for acionado, então o resultado do ou-exclusivo do primeiro passo deve ser passado para a posição zero do registrador correspondente.

Finalmente, a saída é definida pelo ou-exclusivo dos *bits* dos registradores nas posições: R1[18], R2[21] e R[22]. Para a fase de iniciação é utilizada a chave de 64 *bits* e passa de *bit* por *bit* fazendo ou-exclusivo da posição 0 de cada registrador com o *bit* da chave. O algoritmo ignora o resultado das primeiras 100 execuções e depois retorna os *bits* do *keystream* a cada 228 ciclos, gerando os 228 *bits* de saída.

2.1.2.2 Algoritmo A5/2

Esse algoritmo foi desenvolvido por motivos de restrições de exportação do A5/1 e tem uma estrutura similar ao A5/1, com algumas exceções, como por exemplo: o A5/2 utiliza 4 registradores. Outra diferença é o fato que o *clocking* dos registradores R1, R2 e R3 são acionados baseados na regra de maioridade do R4. Ou seja, cada registrador é acionado dependendo dos *bits* do registrador R4.

Registrador	*Clocking Bits*	*Tapping Bits*	*Bits* utilizados na função maioridade
R1	8	13, 16, 17, 18	12, 14*[20], 15
R2	10	20, 21	9, 13, 16*
R3	10	7, 20, 21, 22	13*, 16, 18
R4	3, 7, 10	-	-

Tabela 2 – *Bits* importantes do algoritmo A5/2

[20] *Bit* complementar

[21] Disponível e adaptado de: <http://blog.cryptographyengineering.com/2012/02/sattelite-phone-encryption-is-terrible.html>

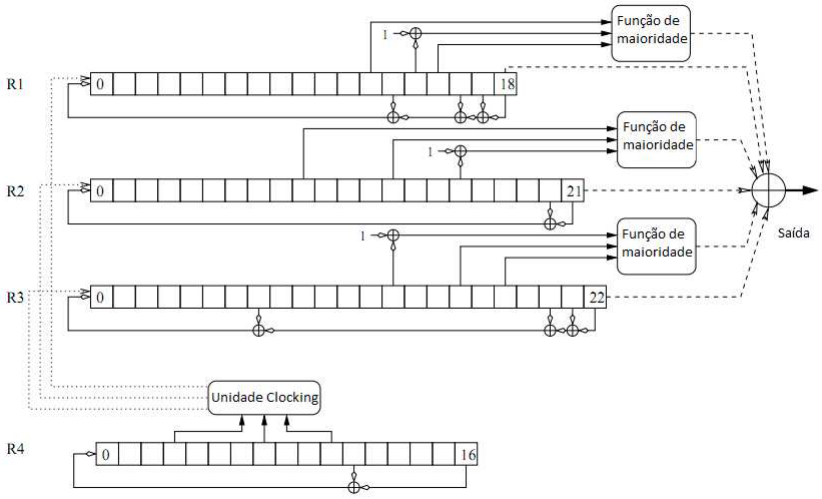

Figura 13 – Estrutura do algoritmo A5/2[21]

Caso o resultado da regra da maioridade dos *clocking bits* do R4 seja igual ao do *bit* que se encontra na posição 3 então o registrador R2 é acionado, caso seja igual ao *bit* que se encontra na posição 7 então o registrador R3 é acionado e se for igual ao *bit* que está na posição 10 então o registrador R1 é acionado. Depois que os registradores são acionados, então o registrador R4 é acionado também. A saída desse algoritmo é feita da forma demonstrada pelo Pseudo-código 2.4

Código 2.4 Pseudo-código A5/2

```
1  void a52Cipher(key key[64]){
2
3    registers R1[18] ,R2[21], R3[22], R4[16]
4
5    clockingBit = marjorityRule(R4[3], R4[7], R4[10])
6
7    resultR1 = 0
8    resultR2 = 0
9    resultR3 = 0
10   if(clockingBit == R4[3]){
11     // R2
12     complement = 1 ^ R2[16]
13     resultR2 = marjorityRule(complement, R2[9], R2[13]) ^ R2[21]
14     deslocateToLeft(R2, R2[21] ^ R2[20])
15
16   } else if(clockingBit == R4[7]) {
17   // R3
18     complement = 1 ^ R3[13]
```

```
19      resultR3 = marjorityRule(complement, R3[16], R3[18]) ^ R3[22]
20      deslocateToLeft(R3, R3[22] ^ R3[21] ^R3[20])
21
22    } else {
23    //R1
24      complement = 1 ^ R1[14]
25      resultR1 = marjorityRule(complement, R1[12], R1[15]) ^ R1[18]
26      deslocateToLeft(R1.R1[18] ^ R1[17] ^R1[16] ^ R1[13])
27    }
28
29    deslocateToLeft(R4, R4[16] ^ R4[11])
30
31    output = resultR1 ^ resultR2 ^ resultR3
32 }
```

1. É calculado o complementar do *bit* da posição especificada de cada registrador.
2. Em cada registrador é utilizado a regra da maioridade para dois *bits* e o complementar do *bit* calculado anteriormente.
3. É feito um ou-exclusivo do resultado do item anterior com o *bit* mais significativo do registrador de cada registrador.

2.1.2.3 Algoritmo E0

Utilizado para proteger informações que utilizam a tecnologia *bluetooth*. O principal objetivo dessa tecnologia é conectar dois aparelhos para transferência de algum tipo de informação.

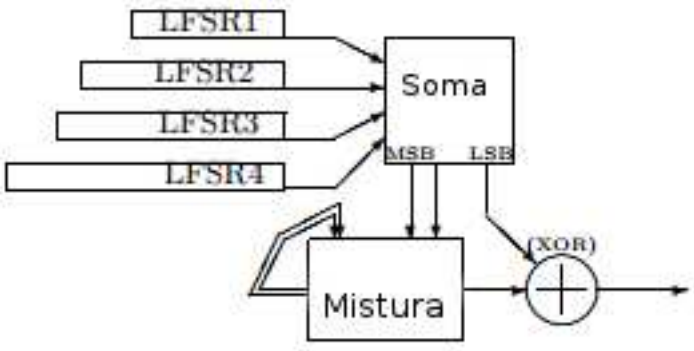

Figura 14 – Estrutura do algoritmo E0(FLUHRER; LUCKS, 2001)

O algoritmo também é baseado no uso de registradores e a saída é gerada em ou-exclusivo de *bits* de cada registrador. Nesta estrutura, são utilizados 4 registradores com

tamanhos 25, 31, 33 e 39 *bits*. O sistema da geração do *keystream* é um pouco diferente, a cada *clock tick* todos os registradores são acionados e é feito um ou-exclusivo da saída de cada um. A operação é descrita a seguir:

Código 2.5 Pseudo-código E0

```
1  void e0Cipher(key key[64]){
2
3    registers R1[25] ,R2[31], R3[33], R4[39]
4
5
6    while(clockTick) {
7      result = sum(R1,R2, R3, R4)
8      char finiteStateMachine[3] = msb(result)
9      output = finiteStateMachine ^ lsb(result)
10   }
11 }
```

2.1.2.4 Algoritmo RC4

Projetado em 1987 por Ron Rivest, é muito utilizado em sistemas de redes como TLS/SSL e WEP por conta de sua simplicidade e alto desempenho. Tem tamanho de chave variável e se baseia em permutações aleatórias. Utiliza um gerador de números aleatórios e as permutações são realizadas sobre esses números aleatórios.

O funcionamento do algoritmo é realizado com uma leitura do texto em claro feita a cada *byte* e cada *byte* produz um *byte* de texto cifrado. Seu funcionamento está ligado a um gerador de *keystream*, ou seja, utilizando como entrada uma chave é criada uma sequência de números aleatórios.

O RC4 é dividido em três passos fundamentais: a inicialização, geração aleatória de *bytes* e cifração.

Inicialização RC4 aceita chaves de tamanho variáveis. Na fase de inicialização, o vetor S é inicializado com valores iniciais seguindo o código mostrado a seguir:

Código 2.6 Código Inicialização

```
1 typedef struct RC4{
2     int i;
3     int j;
4     char state[255];
```

```
5 }
6
7 void rc4Initializer(RC4 * params, char key[], unsigned char *
      temp[255] ){
8
9   int i,j = 0;
10  size_t keylen = strlen(key);
11  for(i = 0; i <= 255; i++){
12    params->state[i] = i;
13    temp[i] = key[i % keylen];
14  }
15  return temp;
16 }
```

Como pode ser visto no código anterior, um vetor temporário de 256 *bytes* é criado juntamente com a inicialização do vetor de 256 *bytes state*. O vetor *temp* é inicializado com os valores da chave K. Se a chave tem tamanho 256 *bytes*, então *temp* é igual a K. Se K for menor que 256 *bytes* então a chave é repetida até preencher as 256 posições de *temp*.

Após a inicialização do vetor *temp*, é feita uma permutação inicial utilizando variáveis de controle i e j e é realizada obedecendo o código abaixo:

Código 2.7 Código Permutação Inicial

```
1 void intialPermutation(RC4 * params, char key[], unsigned
     char * temp[255]){
2
3   int j = 0;
4   unsigned char swap;
5   for(i = 0; i <= 255; i++) {
6     j = (j + params->state[i] + temp[i]) % 256;
7     swap = params->state[j];
8     params->state[j] = params->state[i];
9     params->state[i] = swap;
10  }
11
12  params->i = 0;
13  params->j = 0;
14 }
```

Geração Aleatória de *bytes* Após a permutação inicial, a geração de *bytes* é feita utilizando de permutações aleatórias do vetor *state*. Essa geração utiliza das variáveis

de controle i, j e k. Essa geração é realizada a cada cifração realizada pelo algoritmo. A geração é feita da seguinte forma:

Código 2.8 Código Geração Aleatório de *bytes*

```
1  char rc4Generator(RC4 * params){
2
3    unsigned char t;
4    unsigned char k;
5    unsigned char swap;
6    params->i = (params->i + 1) % 256;
7    params->j = (params->j + params->state[params->i]) % 256;
8    swap = params->state[params->j];
9    params->state[params->j] = params->state[params->i];
10   params->state[params->i] = swap;
11   t = (params->state[params->i] + params->state[params->j]) %
         256;
12   k = params->state[t];
13
14   return k;
15 }
```

Cifração Para a fase de cifração é preciso que se obtenha um *byte* da função de gerador aleatório. A cifração desse algoritmo é feita com a operação ou-exclusivo entre o *byte* proveniente do gerador e o *byte* correspondente do texto em claro.

Código 2.9 Código Cifração de *bytes*

```
1  char rc4Encryption(RC4 * params, char plainText){
2
3     byteToCipher = rc4Generator(params);
4     return plainText ^ byteToCipher;
5  }
```

2.2 Criptografia Assimétrica

A criptografia assimétrica é usada principalmente para cifrar dados pequenos. Um dos principais usos é a assinatura digital e para realizar a troca de chaves de forma segura. Esse tipo de criptografia está associada à criptografia de chave pública. Nela é estabelecido que um par de chaves (chave pública e privada) sejam usadas para a cifração e decifração da mensagem.

O funcionamento é simples. Bob tem suas duas chaves e disponibiliza sua chave pública de forma que qualquer pessoa que tenha interesse em lhe enviar uma mensagem criptografada, utilize sua chave pública. Quando Bob recebe a mensagem cifrada, é realizada a decifração dessa mensagem utilizando sua chave privada.

Alguns dos algoritmos mais conhecidos são o *Elgamal* e o *Diffie-Hellman*. O algoritmo *RSA* foi o primeiro algoritmo a cifrar e decifrar com sucesso utilizando o princípio da chave pública. O *Elgamal* tem sua força proveniente da dificuldade de se calcular um logaritmo discreto. Com o *Elgamal* pode-se realizar a assinatura digital de um documento. *Diffie-Hellman* foram os visionários da chave pública, primeiramente definiram a ideia e algum tempo depois definiram um algoritmo que serve apenas para trocar uma chave secreta entre duas pessoas.

2.2.1 Algoritmo RSA

Tem seu nome baseado nas iniciais dos cientistas do *MIT* que o inventaram: *Ron Rivest, Adi Shamir* e *Len Adleman*. A geração das chaves desse algoritmo é feita utilizando números primos e sua força está associada ao fato de que é fácil realizar a multiplicação de dois números primos grandes, mas é difícil a decomposição desse resultado. Portanto a chave é gerada da seguinte forma:

1. Dois números **p** e **q** são escolhidos de forma aleatória e são primos.
2. $\mathbf{p} \neq \mathbf{q}$
3. **n** é composto pela multiplicação de **p** e **q**.
4. **e**[22], tal que mdc[23](ϕ(n, e)) = 1
5. **d**[24], (e $^{(-1)}$) $\times$ (mod ϕ(n))

A chave pública **e** é usada para cifrar a mensagem e a chave privada **d** é usada para decifrar. O método de cifra é simples, o texto é tratado como um conjunto de inteiros e a cada inteiro é feito o seguinte cálculo:

Cifrar C = M ^{e} mod n. Sendo que M é o inteiro correspondente a parte do texto em claro e n o resultado da multiplicação dos dois números primos selecionados na confecção das chaves, C é o texto cifrado e **e** é a chave pública.

Decifrar M = C ^{d} mod n. Sendo que C é o texto cifrado, M o texto em claro e **d** é a chave privada.

[22] Chave pública, para cifrar um texto.
[23] Operação matemática que visa descobrir o maior divisor comum entre dois ou mais números
[24] Chave privada, para decifrar um texto

Código 2.10 Pseudo-código RSA

```
1  int phi(p, q){
2      return (p - 1) * (q - 1)
3  }
4
5  void rsaCipher(key key[64], char plainText){
6      int p = getBigPrime()
7      int q = getBigPrime()
8
9      while( p == q){
10         q = getBigPrime()
11     }
12
13     int n = p * q
14
15     e = mdc(phi(n, e)
16     while(e != 1){
17         e = mdc(phi(n, e))
18     }
19
20     d = pow(e, (-1)) % (n - 1)
21
22     c = pow(plainText, e) % n
23
24 }
```

2.3 Criptoanálise

Estudo que tem como objetivo a criação de técnicas e métodos que ajudam a obter o texto em claro a partir de um texto cifrado sem ter a devida autorização. Conforme a criptografia cresceu e seus estudos se aprofundaram, os estudos da criptoanálise também cresceram. Abaixo algumas técnicas de criptoanálise.

2.3.1 Força Bruta

Esse ataque visa procurar por exaustão a chave usada para cifrar a mensagem. Para isso testa-se todas as possibilidades de chave e analisa o texto produzido.

É eficiente para chaves pequenas, pois a quantidade de chaves possíveis é pequena. Por se tratar de uma procura pela chave, esse método pode ser aplicado em diferentes

tipos de algoritmo criptográfico.

Tamanho da chave	Tempo para quebrar
32	2.15 milissegundos
56	10.01 horas
128	5.4 X 10^{18} anos
168	5.9 X 10^{30} anos

Tabela 3 – Tempo gasto para encontrar chaves adaptado de (STALLINGS, 2011)

2.3.2 Análise de Frequência

Esse método visa analisar a frequência de ocorrência de cada caractere de um texto cifrado. Existem diversos estudos que indicam a frequência média de ocorrência de cada letra em um texto.

Letra	Frequência no texto	Letra	Frequência no texto
a	8.167	n	6.749
b	1.492	o	7.507
c	2.782	p	1.929
d	4.253	q	0.095
e	12.702	r	5.987
f	2.228	s	6.327
g	2.015	t	9.056
h	6.094	u	2.758
i	6.966	v	0.978
j	0.153	w	2.360
k	0.772	x	0.150
l	4.025	y	1.974
m	2.406	z	0.074

Tabela 4 – Frequência média de letras em um texto em inglês (LEWAND, 2000)

A ideia é identificar as frequências do texto cifrado e substituir as letras a medida que for identificado, até conseguir produzir um texto coeso.

Esse é o método que o algoritmo proposto neste trabalho visa dificultar, visto que sua aplicação pode ser utilizada em diversos tipos de algoritmos criptográficos e seus resultados podem ser satisfatórios. Com o balanceamento das ocorrências do texto cifrado, as frequências médias conhecidas não servirão de apoio para possíveis ataques, visto que cada caractere terá a mesma frequência de ocorrência.

3 Gerador de Números Pseudo-Aleatórios

Há muitos anos a geração de números aleatórios é objeto de estudo da matemática. Por muito tempo, não existiu um algoritmo determinístico para a geração de números que tivesse um período de repetição suficientemente longo para aplicações complexas. Em 1998, os matemáticos japoneses Matishumote e Nishimura inventaram o primeiro algoritmo em que a periodicidade (2^{19937} - 1) excede o número de mudanças rotatórias de elétrons desde a criação do universo (DUTANG; WUERTZ, 2009).

Apesar de existir o *TRNG* ou *True Random Number Generator*, que oferece sequências de números que são aleatórios, as mesmas não são determinísticas e isso faz com que sua utilização seja limitada, visto que em aplicações como a criptografia é necessário reproduzir a mesma sequência para a função de cifrar e de decifrar.

Com isso, a utilização de geradores de números pseudo-aleatórios se faz necessária, porém a sequência gerada deve seguir dois critérios para que seus números possam ser considerados aleatórios:

1 - Aleatoriedade O principal requisito para uma sequência ser considerada aleatória é seus números serem aleatórios. Dois critérios podem ser usados para validar uma sequência:

Distribuição uniforme Os *bits* de uma sequência devem ter frequência aproximada, ou seja, a quantidade de 0 deve ser aproximada da quantidade de 1.

Independência Os números não devem ter relação entre si.

2 - Imprevisibilidade Esse critério serve para garantir que sabendo um número da sequência, é impossível prever o número anterior ou posterior.

Um gerador de números pseudo-aleatórios tem como entrada uma semente.[1] A semente normalmente é gerada a partir de um algoritmo *TRNG*. Pelo fato de serem utilizadas como entrada em algoritmos determinísticos as sementes devem ser guardadas de forma segura.

[1] Termo para especificar um número para geração de uma sequência.

3.1 Geradores

3.1.1 Gerador Linear Congruente

Uma técnica muito usada para geração de números pseudo-aleatórios é um algoritmo que primeiramente foi proposto por *Lehmer*, que é conhecido como método Linear Congruente (STALLINGS, 2011). O algoritmo gera uma sequência dependendo das entradas e a função utilizada para essa geração é:

$$\mathcal{X}_n = (a \times X_{n-1} + c) \mod m \tag{3.1}$$

Para cada número produzido na sequência, essa fórmula é aplicada. A eficiência dessa sequência está diretamente relacionada a escolha das variáveis. A variável **m** é o módulo da equação e a condição de escolha dessa variável é ser maior que 0. A variável **a** é o multiplicador da função e sua condição é ser maior que 0 e menor que **m**. A variável **c** é o incremento e sua condição é estar entre 0 e **m**. O X_0 é a semente para a sequência, será o primeiro elemento da sequência e sua condição é estar entre 0 e **m**.

A escolha dessas entradas determina a periodicidade da sequência e por consequência seu desempenho. Como exemplo de uma sequência com periodicidade 4, as entradas seriam:

Variável	Valor
a	5
c	0
m	16
X_0	1

Tabela 5 – Parâmetros para sequência com periodicidade 4

Portanto o primeiro valor da sequência é o 1 por ser a semente. O segundo valor é calculado da seguinte maneira:

$$\mathcal{X}_n = (5 \times 1 + 0) \mod 16 \tag{3.2}$$

$$\mathcal{X}_n = 5 \tag{3.3}$$

Para o terceiro valor, o cálculo é feito utilizando o resultado do anterior:

$$\mathcal{X}_n = (5 \times 5 + 0) \mod 16 \tag{3.4}$$

$$\mathcal{X}_n = 9 \tag{3.5}$$

Finalmente para o quarto valor, também se utiliza o valor anterior em seu cálculo:

$$\mathcal{X}_n = (5 \times 9 + 0) \mod 16 \tag{3.6}$$

$$\mathcal{X}_n = 13 \tag{3.7}$$

A partir do quinto valor, a sequência se repete:

$$\mathcal{X}_n = (5 \times 13 + 0) \mod 16 \tag{3.8}$$

$$\mathcal{X}_n = 1 \tag{3.9}$$

Ou seja, a escolha dos parâmetros é um passo extremamente importante para esse gerador. Um exemplo com uma periodicidade relativamente grande pode ser obtida trocando apenas o valor de **m** da resolução anterior:

Variável	Valor
a	5
c	0
m	37
X_0	1

Tabela 6 – Parâmetros substituindo o valor de **m**

A sequência gerada a partir dessas variáveis, é:

[1, 5, 25, 14, 33, 17, 11, 18, 16, 6, 30, 2, 10, 13, 28, 29, 34, 22, 36, 32, 12, 23, 4, 20, 26, 19, 21, 31, 7, 25, 24, 9, 8, 3, 15]

Apesar de ter sido maior que a sequência anterior, isso não necessariamente prova sua eficiência, visto que, para aplicações como a criptografia, números muito maiores seriam utilizados e sua periodicidade teria de ser maior também.

3.1.2 Gerador Blum Blum Shub

Uma estratégia para se gerar números pseudo-aleatórios seguramente é conhecida como *Blum Blum Shub*, nomeada pelos seus desenvolvedores. Tem a prova pública de sua força criptográfica de algoritmos para qualquer propósito (STALLINGS, 2011). Os primeiros passos para o algoritmo são:

1. Escolha números primos **p** e **q**.

2. **p** e **q** devem ter resto 3 quando divididos por 4.

3. Obtenha **n**, multiplicando **p** e **q**.

4. Escolha **s** tal que **s** é primo relativo de $\mathbf{n}^2$.

Com isso o gerador produzirá uma sequência com as seguintes condições:

$$\mathcal{X}_0 = s^2 \mod n \tag{3.10}$$

$$\mathcal{X}_n = (X_{n-1})^2 \mod n \tag{3.11}$$

$$\mathcal{B}_n = X_n \mod 2 \tag{3.12}$$

Ou seja, os *bits* menos significantes são retirados a cada número produzido.

3.1.3 Gerador Blum Micali

O gerador Blum Micali é um gerador de números pseudo-aleatórios provadamente seguro e foi criado por Manuel Blum, que também participou do desenvolvimento do Blum Blum Shub, e Silvio Micali (AGERBLAD; ANDERSEN,).

O gerador é executado utilizando os seguintes passos iniciais:

1. Escolha um número primo **p**

2. Então o grupo cíclico[3] Z_p = [1, p-1]

3. a é um elemento de Z_p e é o número aleatório resultante

Com esses passos, o algoritmo tem as entradas para que possa ser realizada a geração dos números. A função que de geração é:

$$\mathcal{G}^k \equiv a \bmod p \tag{3.13}$$

Em que **k** é um número inteiro positivo e g um inteiro que representará um grupo de números inteiros aleatórios.

[2] p e q não são fatores de s

[3] Grupo que é gerado por um elemento que e que o grupo finaliza nesse mesmo elemento.

3.1.3.1 Exemplo

Dado que:

1. p = 17
2. Z_p = [1,16]

Então para g = 3 e k = 4: $3^4 \equiv 13 \mod 17$. Ou seja, o número aleatório 13 satisfaz a condição de estar incluso no grupo Z_p e é o número aleatório do grupo 3.

3.2 Segurança comprovada em geradores de números pseudo-aleatórios

Segurança comprovada em criptografia significa que quebrar um algoritmo criptográfico específico é tão difícil quanto resolver um problema específico conhecido por ser difícil (AGERBLAD; ANDERSEN,).

Uma das aplicações de números pseudo-aleatórios seguros é servir de geradores de *keystream* para a cifra de fluxo. O algoritmo nesse trabalho proposto, irá utilizar esse de tipo de gerador de *keystream*.

Existem dois geradores de números pseudo-aleatórios que são publicamente comprovados em segurança: *Blum Blum Shub* e *Blum Micali*.

3.2.1 Segurança do Blum Blum Shub

A segurança do *Blum Blum Shub* é obtida pelo fato de que seu funcionamento se baseia na residualidade quadrática. Esse problema, é relacionado em decidir se **x** é resíduo quadrático nas condições: mdc(x, N) = 1 e que **x** está entre $1 \leq x \leq N$. [4]

Para ser um resíduo quadrático, **x** tem que ter a seguinte propriedade $y^2 \equiv x \mod N$, para um inteiro y. Para esse cálculo, um algoritmo recebe N e x como parâmetros e retorna 1 se for resíduo quadrático e 0 se não for(AGERBLAD; ANDERSEN,).

3.2.2 Segurança do Blum Micali

Sua segurança é provida pelo problema do logaritmo discreto. Esse gerador utiliza da seguinte função:

$$\mathcal{G}^k \equiv a \mod p \tag{3.14}$$

Realizar a exponenciação logarítmica é um problema fácil de resolver, porém o inverso, o logaritmo discreto, é uma operação comprovadamente difícil.

[4] N é igual a multiplicação dos números primos escolhidos para a geração dos números.

3.2.3 Comparação do Blum Blum Shub e Blum Micali

No trabalho de Josefin Agerblad e Martin Andersen em *Provably Secure Pseudo-Random Generators*, foi realizada uma comparação dos dois geradores de números pseudo aleatórios, que são comprovadamente seguros, nos quesitos de aplicação, velocidade e segurança.

No quesito aplicação, o *Blum Blum Shub* foi considerado mais vantajoso pelo fato que esse gerador é melhor aplicado para cifração de chave pública e o *Blum Micali* ser mais vantajoso para cifrações de chave privada. Como a cifração de chave pública é mais utilizada na criptografia, o *Blum Blum Shub* foi ganhador nesse quesito.

No quesito velocidade, os autores não conseguiram determinar qual gerador é mais rápido, visto que para um gerador ser considerado seguro, o desempenho do mesmo é prejudicado já que suas operações são mais complexas.

No quesito segurança, os autores também não conseguiram chegar a um consenso, visto que o problema que dificulta a quebra dos dois geradores são distintos e igualmente difíceis de serem resolvidos.

4 Proposta de Algoritmo

Neste capítulo será apresentado o algoritmo que este trabalho visa propor, detalhes de seu funcionamento e suas vantagens.

4.1 Motivação

A evolução no campo da criptografia é constante e há sempre novas propostas de algoritmos que focam em solucionar um obstáculo imposto por possíveis atacantes e seus métodos para se obter informações sigilosas.

O algoritmo proposto neste trabalho visa diminuir as possibilidades de análises de frequência de um texto cifrado. É uma nova modalidade na produção de texto cifrado utilizando cifra de fluxo.

4.2 Funcionamento

O algoritmo de cifração terá como entrada um texto em claro e uma sequência de números pseudo-aleatórios e terá como objetivo produzir um texto cifrado onde as frequências de ocorrência dos valores de *bytes* cifrados seja totalmente balanceada.

O mecanismo utilizado para este balanceamento será produzir o texto cifrado em "rodadas" de 256 *bytes* em que cada valor de *byte* cifrado só aparece uma vez por rodada. Isto é, os 256 *bytes* cifrados de cada rodada são distintos, porém a ordem em que aparecem em cada rodada é aleatória (ou melhor dizendo, pseudo-aleatória).

Para se garantir que nenhum valor de *byte* cifrado se repita em uma rodada, utiliza-se uma tabela de 256 posições. No início da rodada a tabela tem todas a suas 256 posições marcadas como não ocupadas. Para cada *byte* de texto em claro que se deseja cifrar será feito um ou-exclusivo com o valor correspondente fornecido pelo gerador de números aleatórios resultando no *byte* pré-cifrado. Tendo-se obtido o *byte* pré-cifrado verifica-se se este valor já está ocupado na tabela.

Caso o valor do *byte* pré-cifrado obtido corresponda a uma posição já ocupada na tabela,temos um conflito. Este valor não será aceito como *byte* cifrado e o próximo valor da sequência pseudo-aleatória será utilizado para tentar novamente a cifração do *byte* em claro. Este procedimento será repetido até que o valor pré-cifrado obtido corresponda a uma posição não ocupada na tabela, quando então este valor será aceito como o *byte* cifrado (correspondente ao byte de texto em claro que está sendo processado pelo cifrador).

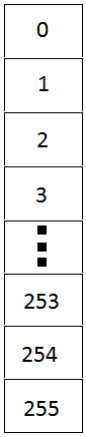

Figura 15 – Tabela de *byte* para uso do algoritmo

O número de valores da sequência pseudo-aleatória não utilizados será contabilizado em uma variável j.

Caso o valor do *byte* pré-cifrado obtido corresponda a uma posição não ocupada na tabela, este valor será aceito como *byte* cifrado, a posição correspondente na tabela será marcada como ocupada, e a variável j será reinicializada com o valor 0.

Aqui se introduz o problema de como enviar ao decifrador as seguintes informações:

1. Que houve um conflito na cifração (quando o mesmo ocorrer)
2. Em ocorrendo o conflito, qual o número de valores na sequência pseudo-aleatória deverão ser desprezados na decifração.

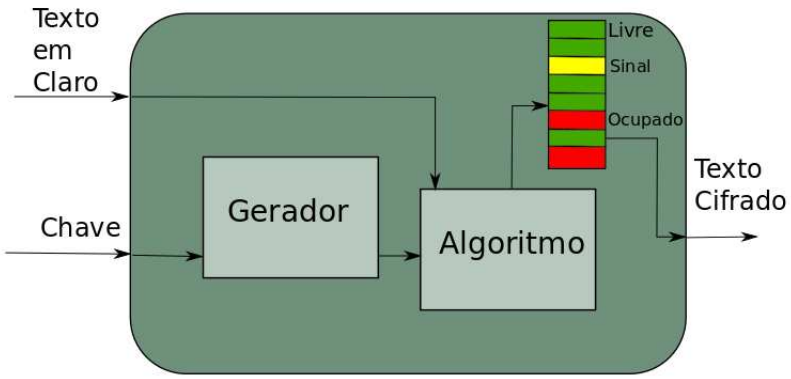

Figura 16 – Esquema do algoritmo de cifração

Para informar ao decifrador que houve um conflito na cifração, propomos a utilização de um valor de *byte* entre 0 e 255 chamado de **sinal**.Este valor é o primeiro *byte* do gerador de *bytes*, e será atualizado a cada vez que a tabela se completa, se tornando o primeiro valor do gerador pseudo-aleatório da próxima rodada. Este valor será enviado pelo cifrador como parte da sequência de *bytes* cifrados indicando ao decifrador que o *byte* seguinte na sequência de *bytes* cifrados corresponderá ao número de valores da sequência produzida pelo gerador pseudo-aleatório que deverão ser desprezados. O decifrador saberá o *byte* recebido é sinal por meio de um contador, que será incrementado a cada vez que um *byte* for decifrado. Após o envio do *byte* de sinal, o cifrador enviará o valor da variável j (também cifrado), e finalmente enviará o *byte* cifrado.

O processo de cifração, que pode ser visto no Pseudo-código 4.1, segue os seguintes caminhos:

O caminho ideal é o que o resultado da operação ou-exclusivo entre o *byte* do texto em claro e o número aleatório correspondente sempre corresponda a uma posição livre na tabela de balanceamento.

Código 4.1 Pseudo-Código Cifração

```
1  void cipher(char plainText, keystream S){
2    unsigned char signal = S[0]; // primeiro sinal do gerador
3    char table[255];
4    int count = 0;
5    int k = 0;
6
7    for(int i = 0; i < 255; i++){
8      table[i] = '0';
9    }
10
11   while(plainText != EOF){
12     if(table[plainText[k] ^ S[k]] != '1'){
13         if(j == 0)
14             output(plainText[k] ^ S[k]);
15         if( j > 0){
16             output(S[old] ^ j);
17             output(S[k+j] ^ plainText[k]);
18             j = 0;
19         }
20         count++;
21     }else if(table[plainText[k] ^ S[k] == '1'){
22         if(j == 0){
```

```
23              output(signal);
24              S[old] = S[k+1];
25              j = 1;
26          }
27          if(j > 0){
28              j = j + 1;
29          }
30      }
31      //se tabela completar
32      if(count == 256){
33          resetTable();
34          signal = S[k]
35          count = 0;
36      }
37      k++;
38    }
39 }
```

Caso se detecte um conflito de resultados na tabela de balanceamento, é enviado o *byte* de sinal para o receptor da mensagem, pois com isso o receptor saberá que o próximo valor recebido será a quantidade de números pseudo-aleatórios que deverão ser descartados da sequência.

Então o algoritmo vai executar o comando de descartar os números pseudo-aleatórios, até que se encontre um resultado em uma posição que esteja livre na tabela de balanceamento.

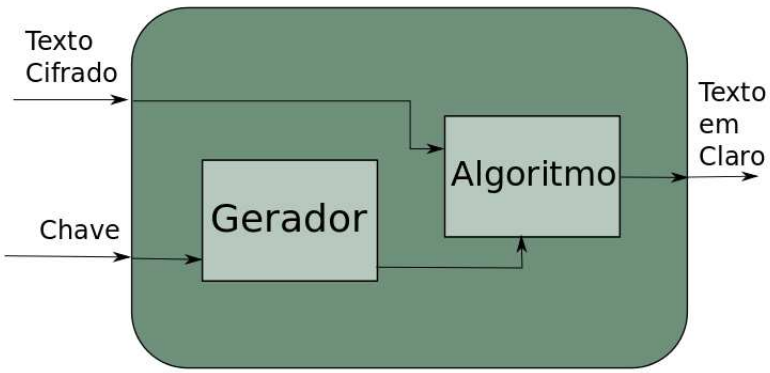

Figura 17 – Esquema do algoritmo de decifração

Ao encontrar essa posição livre, o algoritmo (no lado da cifração) enviará o *byte* de sinal (sem cifrar) e em seguida enviará o resultado do ou-exclusivo entre **j** (quantidade de números descartados) e S_{old}. Então o valor de **j** é novamente reinicializado como 0.

Quando a tabela for totalmente concluída, a rodada termina e o processo se inicia de novo com uma nova rodada, reinicializando-se a tabela (todas as suas posições são marcadas como desocupadas) e um novo sinal é gerado. Esse processo termina quando os *bytes* do texto em claro forem todos cifrados. A cada rodada, o *byte* de sinal terá um valor diferente. Escolhemos o primeiro *byte* da sequência de números pseudo-aleatórios da rodada como o *byte* de sinal.

Podemos observar o processo de decifração no Pseudo-código 4.2. Neste, não é necessário se utilizar a tabela de balanceamento. Para se realizar a decifração é necessário que o decifrador tenha a capacidade de produzir a mesma sequência de números aleatórios que o cifrador. Isto implica que ele deve ter a mesma chave que foi utilizada no processo de cifrar.

No início de cada rodada na decifração, o primeiro *byte* da sequência aleatória é separado como o *byte* de sinal. Os valores pseudo-aleatórios seguintes serão usados no processo de decifração. O primeiro passo no processo de decifração é comparar o *byte* cifrado recebido com o valor de sinal utilizado na rodada.

Código 4.2 Pseudo-Código Decifração

```
1  void decipher(char cipherText, keystream S)
2    char signal = S[0] // primeiro sinal do gerador
3    int count = 0;
4    int k = 0;
5
6    while(cipherText != EOF){
7      if(cipherText[k] != signal){
8          output(cipherText[k] ^ S[k];
9          count++;
10     }
11     if((cipherText[k] == signal){
12         j = cipherText[k+1] ^ S[k+1];
13         S = S[k+j];
14         output(cipherText[k] ^ S);
15         count++;
16     }
17     //se tabela completar
18     if(count == 256){
19         resetTable();
```

```
20          signal = S[k]
21          count = 0;
22      }
23      k++;
24    }
25 }
```

Se o valor do *byte* cifrado for diferente do valor do sinal utilizado na rodada, realiza-se a operação ou-exclusivo entre o *byte* cifrado e o valor fornecido pelo gerador pseudo-aleatório, obtendo-se assim o *byte* de texto em claro correspondente.

Se o valor do *byte* cifrado for igual ao *byte* de sinal utilizado na rodada, isso quer dizer que o próximo *byte* do texto cifrado conterá o valor da variável **j** enviado pelo cifrador, que indica quantos números aleatórios devem ser descartados para o processo de decifração. Ao se descartar esses números, realiza-se a decifração do *byte* cifrado e o processo de decifração volta ao curso normal de execução.

4.3 Vantagens

A maior vantagem deste algoritmo é a equalização das frequências de ocorrência de cada valor de texto cifrado enviado. Excluindo-se os valores dos *bytes* de sinal e os valores dos números de sequência pseudo-aleatórios a serem desprezados na decifração, a frequência de cada valor possível dos *bytes* de texto cifrado será idêntica. Isto é, o algoritmo produz um texto cifrado perfeitamente balanceado.

Outra vantagem do algoritmo de balanceamento de frequência proposto é que ele é totalmente genérico. Isto é, ele funciona com qualquer gerador de números pseudo-aleatórios e algoritmos de cifra de fluxo, tais como: A5/1, A5/2, E0, RC4, *Blum Blum Shub*, *Blum Micali*, etc.

5 Implementação

Neste capítulo serão apresentados detalhes da implementação deste trabalho. Será apresentada a arquitetura que foi criada para o desenvolvimento do algoritmo proposto, um diagrama de sequência, que proverá melhor visibilidade sobre os passos que são seguidos na sua execução e uma explicação sobre os elementos que o compõe.

5.1 Arquitetura

A Figura 18 apresenta a arquitetura que foi utilizada para a realização dos experimentos. A arquitetura foi desenvolvida com o intuito de reproduzir uma aplicação real de um algoritmo de criptografia. As partes do cifrador e do decifrador são executadas em dois computadores distintos que se comunicam utilizando o protocolo de rede TCP/IP.

De maneira geral, cada lado terá uma *thread* que auxiliará no envio/recebimento do *byte* cifrado usando o *socket* TCP/IP. A parte de cifração/decifração será utilizando o núcleo do *RC4* alterado de forma a implementar o algoritmo de balanceamento proposto

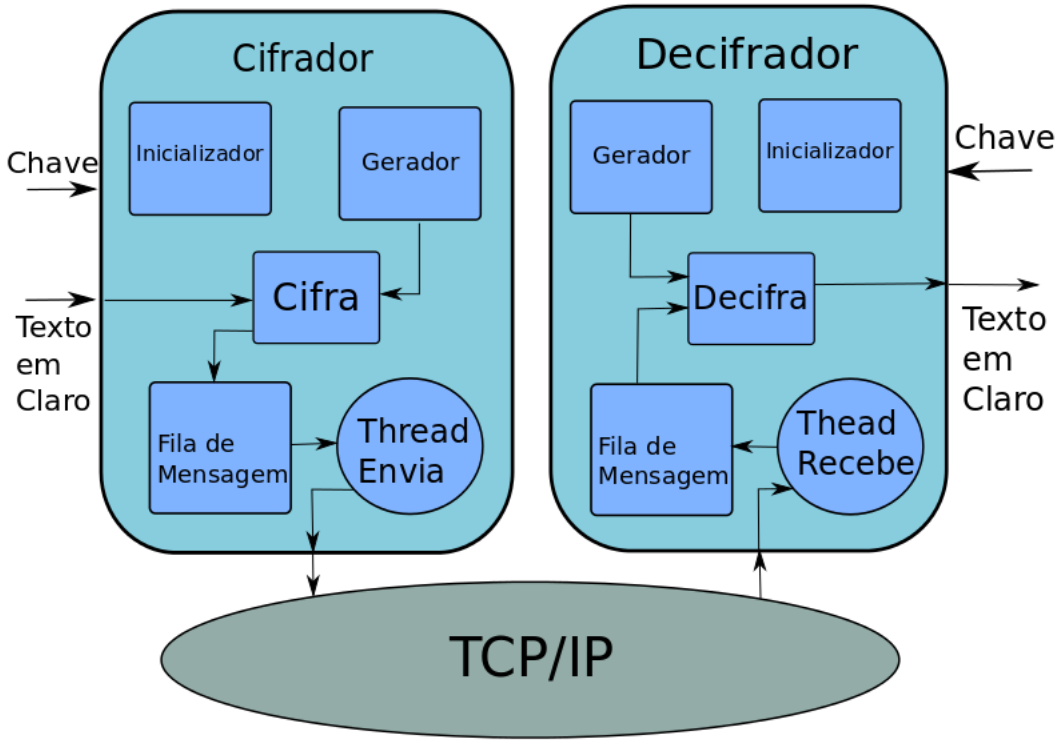

Figura 18 – Arquitetura utilizada no projeto

5.1.1 Inicializador do Gerador

Esse elemento é responsável por fazer a inicialização do gerador que será utilizado no algoritmo.

O gerador escolhido para os experimentos foi o do próprio *RC4*. O *RC4* tem como entradas o texto em claro que será cifrado e uma chave, esta chave é utilizada para fazer a inicialização do gerador. A chave utilizada foi gerada pelo *ssh-keygen*.

5.1.2 Gerador de *Bytes*

Como explicado anteriormente, o gerador escolhido foi o do próprio *RC4* e sua geração foi feita de acordo com o que o algoritmo aconselha.

5.1.3 Fila de Mensagem

A fila de mensagem foi utilizada para servir de intermediário entre o cifrador/decifrador e a *thread* que envia/recebe os *bytes* cifrados.

5.1.4 *Thread* de Envio para Decifrador

Esta thread é responsável por retirar os bytes cifrados da fila de mensagens e os enviar, via *TCP/IP*, para o decifrador.

Optou-se pelo uso desta *thread* para minimizar o *overhead* do envio dos bytes pela rede através da paralelização da execução, evitando, com isso, que o cifrador fique parado aguardando a rede.

5.1.5 *Thread* para Receber do Cifrador

Assim como no cifrador, o decifrador também conta com uma *thread* auxiliar para receber os *bytes* que são enviados pelo cifrador e escrever na fila de mensagem.

Neste caso, provavelmente, o uso da *thread* não traz um ganho apreciável, visto que a rede é o maior gargalo, por isso, a leitura do *socket* é feita a medida que o *buffer* do *TCP/IP* se completa.

5.1.6 *TCP/IP*

Quando estávamos analisando o melhor protocolo de comunicação para ser utilizado em conjunto com o algoritmo, decidimos pelo TCP/IP pela reciprocidade que o mesmo contém„ além da garantia de entrega e sequência dos dados

Em algoritmos de criptografia, a ordem com que os *bytes* são enviados/lidos é de extrema importância, para que o texto decifrado tenha consistência e integridade.

5.1.7 Decifrador

Processo responsável em fazer a tradução dos *bytes* cifrados.

O decifrador irá ter acesso aos *bytes* cifrados na fila de mensagem que será populada pela *thread* de recebimento. A operação que é efetuada é somente um ou-exclusivo do *byte* proveniente do gerador e o *byte* correspondente.

5.1.8 Cifrador

Processo responsável em cifrar os *bytes* em claro que se deseja obter segurança.

Uma vez cifrado, postará na fila de mensagens para que a *thread* de envio faça a sua parte e envie o mesmo para o decifrador

5.2 Diagrama de Sequência

O diagrama de sequência representado na Figura 19 tem como objetivo demonstrar como os processos,cifra e decifra, interagem com os outros elementos, tais como: fila de mensagem, *threads* e a comunicação entre os mesmos.

1. O processo de decifração deve ser o primeiro a ser iniciado, pois ele é responsável por iniciar o servidor em que o processo da cifra irá se conectar para a comunicação e envio de *bytes* cifrados.

2. Em seguida, o processo de decifração irá iniciar a *thread* que será responsável por ler as mensagens proveniente do *socket* de comunicação.

3. O processo de decifração também inicia a fila de mensagem que irá utilizar.

4. Quando o processo de cifração inicia, ele tenta se conectar com o servidor e o servidor o aceita ou não.

5. Quando a comunicação é estabelecida, o processo de cifração inicia a *thread* que será responsável por enviar os *bytes* cifrados para o decifrador.

6. Por fim, o processo de cifração também inicia a fila de mensagem.

7. Os passos anteriores são somente para inicialização dos processos e *threads*. A parte de cifração começa quando tudo já está iniciado e acontece da seguinte forma:

    a) O processo de cifração faz a leitura do arquivo que contém o texto em claro e junto com o *byte* do gerador executa a operação de ou-exclusivo.

    b) Em seguida, o *byte* cifrado é postado na fila de mensagem.

c) Por sua vez, a *thread* que envia os *bytes* cifrados, faz a leitura da fila de mensagem e envia para o decifrador o *byte* cifrado.

d) A *thread* que faz a leitura do *socket*, recebe o valor do *byte* cifrado e o escreve na fila de mensagem do decifrador.

e) Por fim, o decifrador faz a leitura da fila de mensagem e o decifra realizando a operação ou-exclusivo com o *byte* do gerador.

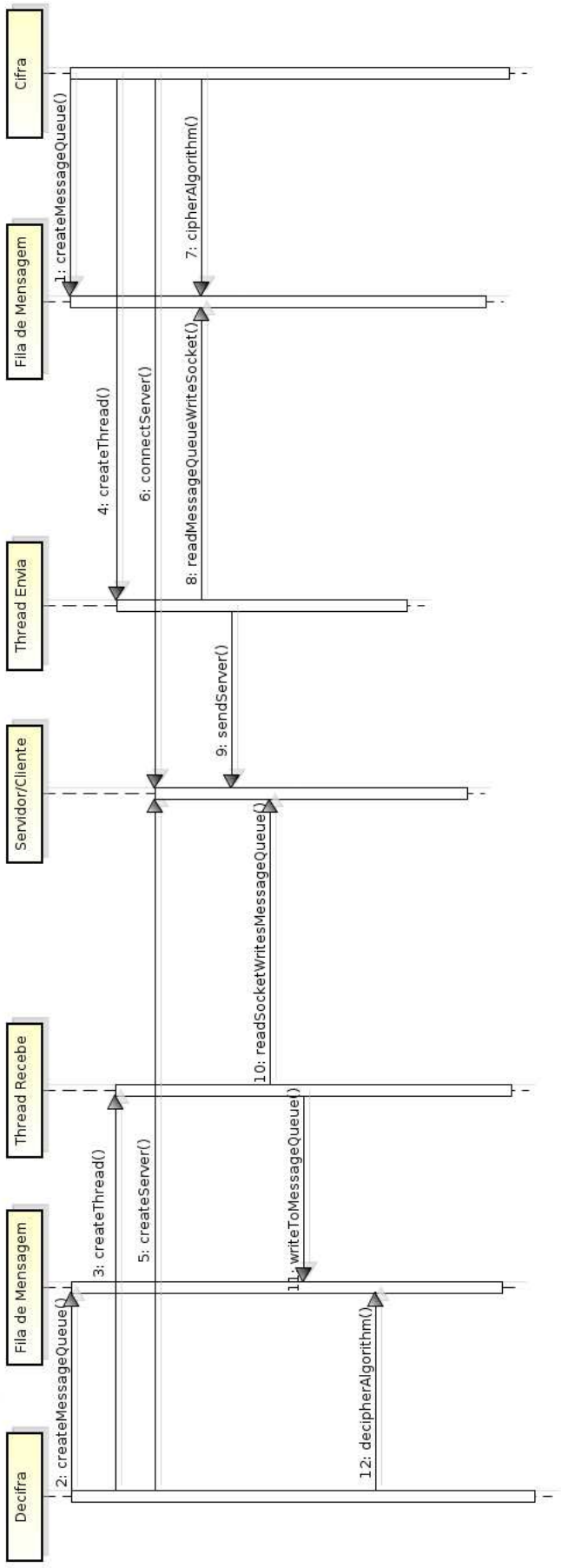

Figura 19 – Diagrama de Sequência

6 Resultados

Neste trabalho foram desenvolvidos diversos experimentos para se chegar a um resultado considerado satisfatório para a implementação do algoritmo proposto.O critério utilizado para a avaliação e validação dos experimentos foi o tempo total de execução. O tempo total de execução é a soma do processo da cifração combinada com a *thread* de envio e a decifração.

Os primeiros experimentos analisaram os tempos do processo de cifração e *thread* de envio e a medição dos tempos médios necessários para a resolução dos conflitos.

Depois de alguns experimentos com os geradores: linear congruente, *Blum Blum Shub* e *Blum Micali*, chegou-se a conclusão que a utilização destes geradores complicaria a análise de desempenho entre o *RC4* e o algoritmo aqui proposto. Isso se deve ao fato de que esses geradores são usados principalmente para geração de números, sem a preocupação com a sua repetibilidade. Para conseguir números com uma repetibilidade menor, seria necessário estudar o uso da chave fornecida para a cifração e transformá-la em entradas satisfatórias para estes geradores. Estes geradores demandam valores muito específicos para que o algoritmo tenha resultados satisfatórios e se esses requisitos não forem satisfeitos pode-se chegar a um *loop* infinito na procura de um *byte* que satisfaça as condições do algoritmo de balanceamento proposto.

O *RC4*, por outro lado, se demonstrou bastante eficaz na geração de números esparsos o suficiente para minimizar os conflitos e evitar a possibilidade do *loop* infinito, sendo, por esta razão, o único utilizado para este estudo.

Uma vez escolhida a melhor implementação do algoritmo, usou-se o *RC4* puro para servir de base de tempo para verificar o *overhead* imposto na cifrarão pelo algoritmo proposto.

6.1 Ambiente Utilizado para Experimentos

Foram utilizados dois computadores para os experimentos aqui descritos. O primeiro computador, que foi o responsável pela cifração do texto em claro, tem processador Intel i7, 16GB de memória RAM e utiliza o sistema operacional Ubuntu 14.04. O segundo computador, responsável pela decifração, tem processador Intel i7, 8GB de memória RAM e também utiliza Ubuntu 14.04 como sistema operacional. O compilador utilizado para ambos os computadores foi o *gcc* 4.8.2.

Durante os experimentos, cabos de rede ligados diretamente no *switch* de rede foram utilizados para que interferências da rede *wireless* não prejudicassem os resultados

dos experimentos.

Como existem fatores externos aos experimentos(internet, por exemplo) os resultados tendem a variar alguns milissegundos. Para se obter o valor mais acurado, foram coletados dez resultados por tamanho de texto para cada algoritmo em cada experimento e calculada a média, que será apresentada nas tabela de resultados dos experimentos

6.2 Tempos de Execução

Esses experimentos foram realizados principalmente para a aproximação de um resultado satisfatório. O fato de se ter tratamento de conflitos faz com que o algoritmo fique com o desempenho inferior ao do *RC4*, porém alguns ajustes foram feitos na arquitetura que foi planejada para a implementação do algoritmo para um resultado mais próximo possível.

6.2.1 Experimento 1

O primeiro experimento foi realizado utilizando somente a arquitetura que foi explanada no Capítulo 5, sem qualquer intervenção sobre o algoritmo. Os resultados foram totalmente insatisfatórios, como podem ser observados na Tabela 7.

Tamanho do	RC4(ms)		RC4 + algoritmo(ms)	
texto(caracteres)	Cifração	Cifração + *Thread*	Cifração	Cifração + *Thread*
10000	12.3	13.4	18.75	19.75
50000	51.6	54.2	97.875	98.875
100000	95.3	96.3	172.125	173.125
500000	1779.2	1836.6	3737.75	3740.5
1000000	6890.3	7023.6	9318.5	9325.75
5000000	48861.6	48905.4	54068	54083.625

Tabela 7 – Tempos obtidos no experimento 1

6.2.2 Experimento 2

Depois de análise dos resultados do primeiro experimento e a comparação dos resultados do *RC4* e *RC4* em conjunto com o algoritmo proposto, percebeu-se que algo que estava sendo utilizado na arquitetura do algoritmo e que não era o algoritmo em si estava afetando o desempenho da execução. A constatação se deve ao fato de que o *RC4* também ter obtido resultados insatisfatórios. Depois de uma análise, descobriu-se que o tamanho da fila de mensagem *default*, que estava sendo utilizado, era o problema. Devido ao seu tamanho estar limitado a poucos caracteres, a *thread* de envio não estava sendo suficientemente rápida e isso estava fazendo com que o tempo da cifração também fosse

prejudicado. Outro indício que esse era o problema e não o algoritmo em si é que o tempo de cifração e tempo da *thread* de envio era praticamente o mesmo.

Com isso, foi efetuado o aumento da fila de mensagem e vimos que com a arquitetura implementada nesse trabalho isso é de extrema importância para o bom desempenho do algoritmo. Os resultados com a fila de mensagem expandida está na Tabela 8.

Tamanho do	RC4(ms)		RC4 + algoritmo(ms)	
texto(caracteres)	Cifração	Cifração + Thread	Cifração	Cifração + Thread
10000	6.2	10.4	11.8	17.4
50000	22.6	34.4	40	60.6
100000	36.2	60	73	114.2
500000	154.4	279.6	333.6	548.2
1000000	304.2	544.2	663.2	1169.6
5000000	1400.4	3528	3139	6743.2

Tabela 8 – Tempos obtidos no experimento 2

6.2.3 Experimento 3

Com os resultados obtidos pelo experimento anterior, resolvemos reavaliar a necessidade de enviar *byte* a *byte* o texto cifrado. Isso porque depois de algumas análises, percebemos que a quantidade de conflitos em uma cifração era equivalente a metade do texto em claro e para cada conflito dois *bytes* a mais eram enviados, ou seja, a quantidade de bytes a mais enviados pelo algoritmo proposto era:

$$tcifrado = tclaro + 2 \times (tclaro/2) \tag{6.1}$$

Isso resulta no dobro do texto em claro e que somente o algoritmo proposto envia. Portanto, para diminuir a quantidade de envios, decidimos enviar os três *bytes* referentes ao conflito(sinal, quantidade de *bytes* descartados e *byte* cifrado) de uma só vez , aproximando, com isso, da quantidade de envios que o *RC4* realiza. O resultado desse experimento está descrito na Tabela 9.

Tamanho do	RC4(ms)		RC4+Algoritmo(ms)	
texto(caracteres)	Cifração	Cifração + *Thread*	Cifração	Cifração + *Thread*
10.000	7.8	15.1	12.6	15.9
50.000	19.3	33.6	28.2	34.9
100.000	35.9	60.1	50.5	61
500.000	157	279.2	207.2	427.5
1.000.000	302.8	548.6	397.6	1020.4

Tabela 9 – Tempos obtidos no experimento 3

6.2.3.1 Tempo Total de Execução

Como os resultados anteriores foram os mais satisfatórios que obtivemos durante toda a fase de testes desse trabalho, decidimos dar uma visão de tempo de execução total. Nessa parte do experimento, o tempo de execução que foi levado em conta foram: cifração, *thread* de envio e recebimento e decifração. Para se obter o tempo total foi necessário a utilização de um meio de informar para o lado da decifração que o lado da cifração havia acabado,porque senão a decifração ficaria esperando novos *bytes* para decifrar. Portanto duas opções foram testadas:

Envio de dois *bytes* Nesse experimento, para cada *byte* cifrado enviado era enviado um *byte* que indicava o final do texto em claro para o lado da decifração.

Tamanho do texto(carater)	RC4(ms)	RC4 + algoritmo(ms)	Tempo a mais(ms)	Porcentagem
10000	106.8	108.1	1.32	1.24
50000	517.4	530	12.56	2.43
100000	1036	1061.6	25.6	2.47
500000	5152.6	5306.6	153.93	2.99
1000000	10321.1	10607.1	285.99	2.77
5000000	51591.8	53276.7	1684.92	3.27

Tabela 10 – Experimento tempo total enviando 2 *bytes*

Essa opção de envio foi descartada, visto que o tempo que a *thread* utiliza para enviar aumentou consideravelmente. Portanto, decidimos pela estratégia que mostraremos a seguir.

Envio de sequência de caracteres Nesse experimento, ao final do texto em claro foi enviado uma sequência de três bytes 0 como um marcador de fim de cifragem.

Tamanho do texto(caracteres)	RC4(ms)	RC4+Algoritmo(ms)	Tempo a mais(ms)	Porcentagem
10.000	125.2	127.6	2.4	1.92
50.000	605.9	623	17.1	2.82
100.000	1144.8	1214.6	69.8	6.09
500.000	5730	6042.2	312.2	5.45
1.000.000	11,429.6	12,048.7	619.1	5.42

Tabela 11 – Tempo enviando sinal de fim do arquivo '000'

Com esses resultados obtivemos em média 4,34% de tempo a mais. A maior porcentagem obtida foi com o texto de tamanho 100.000 caracteres, 6,09%.

6.2.4 Resolução de Conflitos

Este experimento, visa dar maior visibilidade sobre a quantidade média de conflitos que ocorrem em diversos tamanhos de textos e o tempo médio das resoluções. Esses tempos podem ter uma variação, visto que para cada conflito a função de obter o tempo do sistema foi utilizada duas vezes. Para textos pequenos, a variação pode ser pequena, porém a medida que o texto fica maior, esse tempo tende a aumentar também. A Tabela 12 mostra os resultados obtidos.

Tamanho do texto(caracteres)	Tempo Extra(ms)	Quantidade de Conflitos	Tempo médio de resolução(ms)
10.000	10.4	5.037	0.002065
50.000	32.8	25.174	0.001303
100.000	67	49.998	0.001340
500.000	313.7	250.086	0.001254
1.000.000	611.8	499.890	0.001224
5.000.000	3029.1	2.497.605	0.001213

Tabela 12 – Quantidade de conflitos e tempo médio de resolução.

Percebe-se que a média de conflitos para cada tamanho de texto é aproximadamente a metade do tamanho do texto em claro. Outra percepção que pode ser obtida pela tabela, é que conforme a quantidade de conflitos aumenta, o tempo médio de resolução diminui.

6.3 Resultados de Vetores de Testes

Tendo como objetivo garantir que a minha implementação do algoritmo *RC4* está correta, utilizei os testes citados por STROMBERGSON; JOSEFSSON na *RFC6269*. Neste documento, no Anexo A, temos diferentes tamanhos de chaves e o *keystream* esperado. O resultado gerado pela implementação deste trabalho está em conformidade com os resultados apresentados no *RFC* acima citado.

7 Conclusão

Conforme visto ao longo deste trabalho, existem formas de se obter informações, que devem ser mantidas em sigilo, mesmo não tendo a devida autorização e isto faz com que haja uma insegurança, por parte dos usuários dos algoritmos criptográficos, em utilizá-los. Visando resolver esse problema, a proposta deste trabalho é fazer a definição de um algoritmo de cifra de fluxo com balanceamento de frequência de ocorrência de caracteres.

Existem diversas técnicas de criptoanálise. Uma dessas é a análise de frequência de ocorrência de caracteres de um texto cifrado. Com essa técnica, o atacante pode ter acesso ao texto em claro, utilizando referências de frequência de ocorrência média de caracteres para cada idioma. O algoritmo, neste trabalho proposto, tem como foco dificultar a utilização dessa técnica, pois as frequências de ocorrência dos caracteres do texto cifrado serão balanceadas e assim torna-se impossível determinar qual caractere terá maior frequência no mesmo.

Outro ataque conhecido como *Statistical Mobius Analysis* tem como hipótese que um sistema criptográfico(tal como um gerador pseudo aleatório) obedece a uma geração de *bytes* com distribuição fixa (FILIOL, 2002). Como o algoritmo proposto neste trabalho não utiliza todos os *bytes* que são produzidos pelo gerador pseudo aleatório(em cada conflito em sua tabela interna alguns *bytes* da sequência pseudo aleatória precisam ser descartados, dependendo do texto em claro que está sendo cifrado) este ataque não funcionaria sobre uma mensagem cifrada utilizando o algoritmo proposto.

Klein (2006) propõe um ataque utilizando uma relação, descoberta por ele, entre a saída do gerador pseudo aleatório e os estados internos do *RC4*. Dada uma distribuição de estados internos uniformes Klein obtém duas sequências possíveis de *bytes* que serão gerados pelo gerador pseudo aleatório. Por essa abordagem é possível determinar todos os *bytes* que serão produzidos pelo *RC4*. O algoritmo proposto neste trabalho também mitiga esse ataque, pois o descarte de *bytes* da sequência pseudo aleatória(que ocorre durante a resolução de conflitos em sua tabela interna) confunde as previsões de um atacante que utiliza a relação de Klein.

Após a execução dos experimentos, podemos afirmar que a utilização do algoritmo em conjunto com o *RC4* é possível com um impacto pequeno no desempenho do tempo total. Algumas observações devem ser feitas, por exemplo:

1. a utilização da fila de mensagem do *Linux* impacta em haver uma alteração em seu tamanho para que o algoritmo funcione sem impactos

2. a utilização desse algoritmo é indicado para comunicações síncronas, pois somente

dessa forma, os desempenhos do *RC4* e do *RC4* e o algoritmo se assemelham.

7.1 Trabalhos Futuros

Comunicações assíncronas Após análises, foi percebido que uma melhoria que deve ser implementada é a possibilidade de se utilizar o algoritmo em comunicações assíncronas.

Multi-*threading* Uma maneira de auxiliar o algoritmo em questão de desempenho, seria implementá-lo de forma a aceitar ser multi-*threading* para a cifração e envio. Uma maneira de fazer com que as mensagens cheguem na ordem deve ser pensada.

Fila de mensagem estudar uma forma de implementar a comunicação entre as threads sem o seu uso para evitar que o seu tamanho seja um gargalo

Geradores Encontrar uma maneira eficiente de utilizar os geradores: linear congruente, *Blum Blum Shub* e *Blum Micali*

Referências

AGERBLAD, J.; ANDERSEN, M. Provably secure pseudo-random generators, a literary study. Citado 2 vezes nas páginas 52 e 53.

BAKHTIARI, M.; MAAREF, M. A. An efficient stream cipher algorithm for data encryption. University Technology Malaysia, 2011. Citado na página 36.

BIRYUKOV, A. Block ciphers and stream ciphers: The state of the art. Katholieke Universiteit Leuven, 2011. Citado na página 31.

COLLINS, C. A comparative analysis of mark 1, colossus and zuse z4. 2006. Citado na página 30.

DAEMEN, J.; RIJMEN, V. Aes proposal: Rijndael. 1998. Citado na página 36.

DAVIES, D. A brief history of cryptography. 1997. Citado na página 29.

DUTANG, C.; WUERTZ, D. A note on random number generation. 2009. Citado na página 49.

FILIOL, E. A new statistical testing for symmetric ciphers and hash functions. 2002. Citado na página 73.

FLUHRER, S. R.; LUCKS, S. Analysis of the e0 encryption system. 2001. Citado 2 vezes nas páginas 15 e 41.

KLEIN, A. Attacks on the rc4 stream cipher. 2006. Citado na página 73.

LEWAND, R. E. *Cryptological Mathematics*. [S.l.]: The Matematical Association of America, 2000. Citado na página 47.

MAXIMOV, A. *Some Words on Cryptanalysis of Stream Ciphers*. Tese (Doutorado) — Lund University, 2006. Citado na página 27.

STALLINGS, W. *Cryptography and Network Security, Principles and Practice*. [S.l.]: Prentice Hall, 2011. Citado 3 vezes nas páginas 47, 50 e 51.

STROMBERGSON, J.; JOSEFSSON, S. Test vectors for the stream cipher rc4. 2011. Citado na página 71.

VAUDENAY, S. *A Classical Introduction To Cryptography, Applications for Communications Security*. [S.l.: s.n.], 2006. Citado na página 29.

WILCOX, J. Solving the enigma: History of the cryptanalytic bombe. 2006. Citado na página 29.

Apêndices

APÊNDICE A – Códigos

A.1 Bibliotecas

A.1.1 Conexão

```
1 #ifndef CONNECTIONS_H
2 #define CONNECTIONS_H
3
4 #include <functions.h>
5
6 /**
7  * Variables for network tcp
8  */
9 int sockdecipher, sockcipher, portno, clilen;
10 struct sockaddr_in serv_addr, cli_addr;
11 struct hostent *server;
12
13 /**
14  * Create server
15  */
16
17 int createServer(int argc, char *argv[]);
18
19 /**
20  * Connect to decipher(server)
21  */
22
23 int connectServer(int argc, char *argv[]);
24
25 /**
26  * Accept cipher(client)
27  */
28
29 int acceptClient();
30
31 /**
32  * Send byte to decipher(server)
33  */
```

```
34
35 int sendServer(message msg);
36
37 /**
38  * Accept byte from cipher(client)
39  */
40
41 int receiveMessage(unsigned char * cipherByte);
42
43 /**
44  * Close socket
45  **/
46
47 void closeSocket(int sockfd);
48
49 /**
50  * Print error for network
51  */
52
53 void error(char *msg);
54
55 #endif
```

A.1.2 Funções

```
1 #ifndef FUNCTIONS_H
2 #define FUNCTIONS_H
3 /**
4  * Libraries neccessary to the program
5  */
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <sys/types.h>
9 #include <sys/socket.h>
10 #include <netinet/in.h>
11 #include <netdb.h>
12 #include <string.h>
13 #include <unistd.h>
14 #include <pthread.h>
15 #include <sys/ipc.h>
16 #include <sys/msg.h>
17 #define TABLE_SIZE 255
```

```
18 #define CHARACTERS 255
19
20 /**
21  *  Structs for the program
22  **/
23
24 typedef struct RC4 {
25   int i;
26   int j;
27   unsigned char state[256];
28 } RC4;
29
30 typedef struct threadParameters {
31   char * fileOut;
32   char * fileName;
33   int eof;
34 } threadParameters;
35
36 typedef struct message {
37   long type;
38   short eof;
39   short bytes;
40   unsigned char characteres[30];
41   char filler[20];
42 } message;
43
44 /**
45  * Message queue id
46  **/
47
48 static int msqid;
49
50 /**
51  * Global variables for the program
52  */
53
54 char table[TABLE_SIZE];
55 unsigned char signal;
56
57 /**
58  *  Stores child as a thread
59  */
```

```
60
61 static pthread_t readsMessage;
62
63 /**
64  * Reset table
65  */
66
67 void resetTable();
68
69 /**
70  * RC4 Initializer
71  */
72
73 void rc4Initializer();
74
75 /**
76  * RC4 Get new byte
77  */
78
79 char rc4Generator();
80
81 /**
82  * Cipher/Decipher byte
83  */
84
85 unsigned char xorOperation(char textByte, char byteGenerated);
86
87 /**
88  * Checks if table is occupied
89  */
90
91 int checkTable(unsigned char cipherByte);
92
93 /**
94  * Cipher text with algorithm
95  */
96
97 int cipherAlgorithm(RC4 * params, char plaintext);
98
99 /**
100  * Thread to decipher a text with algorithm
101  */
```

```
102
103 void decipherAlgorithm(RC4 * params);
104
105 /**
106  * Rc4 plain cipher algorithm
107  */
108
109 void rc4CipherAlgorithm( RC4 * params,char plaintext);
110
111 /**
112  * Rc4 Plain Decipher Algorithm
113  */
114
115 void rc4DecipherAlgorithm( RC4 * params);
116
117 /**
118  * Creates thread
119  */
120
121 void createThread(threadParameters * dataThread,int type);
122
123 /**
124  * Creates message queue
125  */
126
127 int createMessageQueue(char * fileName, int type);
128
129 /**
130  * Destroy message queue
131  */
132
133 void destroyMessageQueue();
134
135 /**
136  * Writes character to message queue
137  */
138
139 int writeMessageToQueue(message msg, int count);
140
141 #endif
```

A.2 Conexão e Funções

A.2.1 Conexão

```
1 #include <connections.h>
2 #include <functions.h>
3
4 /**
5  * Create server
6  */
7
8 int createServer(int argc, char *argv[]){
9
10   if (argc < 2) {
11     fprintf(stderr,"ERROR, no port provided\n");
12     exit(1);
13   }
14   sockdecipher = socket(AF_INET, SOCK_STREAM, 0);
15   if (sockdecipher < 0)
16     error("ERROR opening socket");
17
18   bzero((char *) &serv_addr, sizeof(serv_addr));
19   portno = atoi(argv[1]);
20   serv_addr.sin_family = AF_INET;
21   serv_addr.sin_addr.s_addr = inet_addr("0.0.0.0");
22   serv_addr.sin_port = htons(portno);
23   if (bind(sockdecipher, (struct sockaddr *) &serv_addr,sizeof(
       serv_addr)) < 0) {
24     error("ERROR on binding");
25     return 1;
26   }
27   return 0;
28
29 }
30
31 /**
32  * Connect to decipher(server)
33  */
34
35 int connectServer(int argc, char *argv[]){
36   if (argc < 3) {
37     fprintf(stderr,"usage %s hostname port\n", argv[0]);
```

```
38      exit(0);
39    }
40    portno = atoi(argv[2]);
41    sockdecipher = socket(AF_INET, SOCK_STREAM, 0);
42    if (sockdecipher < 0)
43      error("ERROR opening socket");
44
45    server = gethostbyname(argv[1]);
46    if (server == NULL) {
47      fprintf(stderr,"ERROR, no such host\n");
48      exit(0);
49    }
50
51    bzero((char *) &serv_addr, sizeof(serv_addr));
52    serv_addr.sin_family = AF_INET;
53    bcopy((char *)server->h_addr,
54    (char *)&serv_addr.sin_addr.s_addr,
55    server->h_length);
56    serv_addr.sin_port = htons(portno);
57    if (connect(sockdecipher,(struct sockaddr *)&serv_addr,sizeof(
          serv_addr)) < 0){
58      error("ERROR connecting");
59      return 1;
60    }
61    return 0;
62 }
63
64 /**
65  * Accept cipher(client)
66  */
67
68 int acceptClient(){
69    listen(sockdecipher,5);
70    clilen = sizeof(cli_addr);
71    sockcipher = accept(sockdecipher, (struct sockaddr *) &cli_addr
          , &clilen);
72    if (sockcipher < 0){
73      error("ERROR on accept");
74      return 1;
75    }
76    return 0;
77 }
```

```
78
79 /**
80  * Close some socket
81  **/
82
83 void closeSocket(int sockfd) {
84   shutdown(sockfd, SHUT_RDWR);
85   close(sockfd);
86 }
87
88 /**
89  * Print error for network
90  */
91
92 void error(char *msg){
93   perror(msg);
94   exit(1);
95 }
96
97 /**
98  * Send byte to decipher(server
99  */
100
101 int sendServer(message msg){
102   int n;
103   n = write(sockdecipher, msg.characteres, msg.bytes);
104   if (n < 0){
105       error("ERROR writing to socket");
106       return 0;
107   }
108   return 1;
109 }
110
111 /**
112  * Accept byte from cipher(client)
113  */
114
115 int receiveMessage(unsigned char * cipherByte){
116   int n;
117   n = read(sockcipher,cipherByte,8192);
118     if (n < 0){
119       error("ERROR reading from socket");
```

```
120       return 1;
121     }
122   return n;
123 }
```

A.2.2 Funções

```
1  #include <functions.h>
2  #include <connections.h>
3  #include <pthread.h>
4  #include <sys/ipc.h>
5  #include <sys/msg.h>
6  #include <arpa/inet.h>
7  #include <netinet/in.h>
8
9  /**
10  * Reset table
11  */
12
13 void resetTable(){
14   int i = 0;
15   for(; i <= TABLE_SIZE; i++){
16     table[i] = '0';
17   }
18 }
19
20 /**
21  * RC4 Initializer
22  */
23
24 void rc4Initializer(RC4 * params, char key[]){
25
26   int i,j = 0;
27   size_t keylen = strlen(key);
28   unsigned char temp[255];
29   for(i = 0; i <= CHARACTERS; i++){
30     params->state[i] = i;
31     temp[i] = key[i % keylen];
32   }
33
34   unsigned char swap;
35   for(i = 0; i <= CHARACTERS; i++) {
```

```
36      j = (j + params->state[i] + temp[i]) % 256;
37      swap = params->state[j];
38      params->state[j] = params->state[i];
39      params->state[i] = swap;
40    }
41
42    params->i = 0;
43    params->j = 0;
44 }
45
46 /**
47  * RC4 Get new byte
48  */
49
50 char rc4Generator(RC4 * params){
51
52    unsigned char t;
53    unsigned char k;
54    unsigned char swap;
55    params->i = (params->i + 1) % 256;
56    params->j = (params->j + params->state[params->i]) % 256;
57    swap = params->state[params->j];
58    params->state[params->j] = params->state[params->i];
59    params->state[params->i] = swap;
60    t = (params->state[params->i] + params->state[params->j]) %
        256;
61    k = params->state[t];
62
63    return k;
64 }
65
66 /**
67  * Cipher/Decipher byte
68  */
69
70 unsigned char xorOperation(char textByte, char byteGenerated){
71
72    return textByte ^ byteGenerated;
73 }
74
75 /**
76  * Checks if table is occupied
```

```
77  */
78
79 int checkTable(unsigned char cipherByte){
80
81   if(table[cipherByte] == '1'){
82     //Value is occupied
83     return 1;
84   } else {
85     //Value is not occupied
86     return 0;
87   }
88 }
89
90 /**
91  * Writes character to message queue
92  */
93
94 int writeMessageToQueue(message msg, int count){
95   // Fix the problem with the queue when the cipher result is 0
96   if(msgsnd(msqid, &msg, 30, 0) == -1)
97   {
98     perror("msg_sender: Error while writing message to queue");
99     exit(1);
100  }
101  return 0;
102 }
103
104
105 /**
106  * Thread to cipher with algorithm
107  */
108
109 int cipherAlgorithm(RC4 * params, char plaintext){
110
111   unsigned char byteToCipher, cipherTextByte, oldS, cipherJ;
112   int result, j = 0, position = 0;
113   int count = 1;
114   message msg;
115   msg.type = 1;
116   msg.eof = 0;
117   do{
118     if(j == 255){
```

```
119       cipherJ = xorOperation(j, oldS);
120       msg.characteres[position] = cipherJ;
121       position++;
122       msg.characteres[position] = signal;
123       position++;
124       oldS = rc4Generator(params);
125       j = 1;
126     }
127     byteToCipher = rc4Generator(params);
128     cipherTextByte = xorOperation(plaintext, byteToCipher);
129
130     result = checkTable(cipherTextByte);
131
132     if(result == 0) {
133       if(j == 0){
134         msg.characteres[position] = cipherTextByte;
135         position++;
136         msg.bytes = position;
137         writeMessageToQueue(msg,position);
138         position = 0;
139
140       }else {
141         // cipher j with old_s and put on ipc along with
              ciphertext
142         cipherJ = xorOperation(j, oldS);
143         msg.characteres[position] = cipherJ;
144         position++;
145         msg.characteres[position] = cipherTextByte;
146         position++;
147         msg.bytes = position;
148         writeMessageToQueue(msg,position);
149         position = 0;
150
151       }
152       table[cipherTextByte] = '1';
153       break;
154     }else{
155       if(j == 0){
156         //put signal on ipc
157         msg.characteres[position] = signal;
158         position++;
159         oldS = byteToCipher;
```

```
160         j = 1;
161       }else{
162         j = j + 1;
163       }
164     }
165   }while(result == 1);
166
167   return 0;
168 }
169
170 /**
171  * Thread to decipher a text with algorithm
172  */
173
174 void decipherAlgorithm(RC4 * params){
175
176   unsigned char byteToDecipher, plainTextByte, cipherTextByte;
177   int j = 0, countTable = 256;
178   int charsRead, n;
179   message msg;
180   msg.type = 1;
181   while(1){
182     if(countTable > 255){
183       signal = rc4Generator(params);
184       countTable = 1;
185     }
186     charsRead = msgrcv(msqid, &msg, 30, 0 , 0);
187     byteToDecipher = rc4Generator(params);
188     if(msg.characteres[0] == signal) {
189       charsRead = msgrcv(msqid, &msg, 30, 0, 0);
190       j = xorOperation(msg.characteres[0],byteToDecipher);
191       int i = 1;
192       for(i = 1; i < j; i++){
193         byteToDecipher = rc4Generator(params);
194       }
195       continue;
196     }
197     plainTextByte = xorOperation(msg.characteres[0],
          byteToDecipher);
198     countTable++;
199     printf("%c", plainTextByte);
200     if(msg.eof == 1){
```

```
201         unsigned char test = '1';
202         n = write(sockcipher, &test, 1);
203         break;
204       }
205     }
206 }
207
208 /**
209  * Rc4 Plain Cipher Algorithm
210  */
211
212 void rc4CipherAlgorithm( RC4 * params,char plaintext){
213
214   unsigned char byteToCipher, cipherTextByte, oldS, cipherJ;
215   int result, j = 0;
216   int count = 1;
217   message msg;
218   msg.type = 1;
219   msg.eof = 0;
220
221   byteToCipher = rc4Generator(params);
222   cipherTextByte = xorOperation(plaintext, byteToCipher);
223
224   //put cipherTextByte on IPC
225   msg.characteres[0] = cipherTextByte;
226   msg.bytes = 1;
227   writeMessageToQueue(msg,msg.bytes);
228
229 }
230
231
232 /**
233  * Rc4 Plain Decipher Algorithm
234  */
235
236 void rc4DecipherAlgorithm( RC4 * params){
237   unsigned char byteToDecipher, plainTextByte;
238   int charsRead, n;
239   message msg;
240   msg.type = 1;
241   msg.eof = 0;
242   while(1){
```

```
243     charsRead = msgrcv(msqid, &msg, 30, 0 , 0);
244
245     if(msg.eof == 1){
246       unsigned char test = '1';
247       n = write(sockcipher, &test, 1);
248       break;
249     }
250     byteToDecipher = rc4Generator(params);
251     plainTextByte = xorOperation(msg.characteres[0],
          byteToDecipher);
252     printf("%c", plainTextByte);
253   }
254 }
255
256 /**
257  * Creates message queue
258  */
259
260 int createMessageQueue(char * fileName, int type){
261     #ifdef PRINT_DEBUG
262         printf("msg_receiver: Initiating\n");
263     #endif
264
265   key_t key;      /* Key to access message queue */
266
267     /* Create a key  */
268     key = ftok(fileName, 'A');
269     if(key == -1)
270     {
271         perror("msg_receiver: Error while creating message key");
272         exit(1);
273       }
274     #ifdef PRINT_DEBUG
275         printf("msg_receiver: message key created: %i\n", key);
276     #endif
277
278     /* With the created key, tries connect/create to a message
          queue */
279     msqid = msgget(key, 0644 | IPC_CREAT);
280     if(msqid == -1)
281     {
282         perror("msg_receiver: Coundn't create or connect to a
```

```
            message queue");
283         exit(1);
284     }
285     #ifdef PRINT_DEBUG
286         printf("msg_receiver: message id created: %i\n", msqid);
287     #endif
288
289   return 0;
290 }
291
292 /**
293  * Destroy message queue
294  */
295
296 void destroyMessageQueue(){
297     /* Destroys message queue */
298     if(msgctl(msqid, IPC_RMID, NULL) == -1)
299     {
300         perror("msgctl");
301         exit(1);
302     }
303 }
304
305
306 /**
307  * Reads from message queue and writes on file
308  */
309
310 static void * readMessageQueueWriteSocket(void * params){
311     threadParameters * data = params;
312   message msg;
313   msg.type = 1;
314   int count = 0;
315   int charsRead;
316
317     while(1){
318       charsRead = msgrcv(msqid, &msg, 30, 0, 0);
319         if(msg.eof == 1){
320           msg.characteres[0] = '0';
321           msg.bytes = 1;
322           for(count = 0; count < 3; count++){
323             sendServer(msg);
```

```
324             }
325             data->eof = 1;
326             break;
327           }
328           count++;
329           sendServer(msg);
330       }
331 }
332
333 /**
334  * Read file and writes on message queue
335  */
336
337 static void * readSocketWritesMessageQueue(void * params){
338   threadParameters * data = params;
339   message msg;
340   msg.type = 1;
341   unsigned char byte[8192];
342   int charsRead;
343   short notFirst = 0;
344   int count = 0;
345   int eof = 0;
346
347   while(1){
348     charsRead = receiveMessage(&byte[0]);
349     if(charsRead == 0){
350       if(notFirst){
351         msg.eof = 1;
352         writeMessageToQueue(msg,count);
353         break;
354       }
355       notFirst = 1;
356       continue;
357     }
358     msg.bytes = 1;
359     for(count = 0; count < charsRead; count++){
360       if(byte[count] == '0'){
361         eof++;
362         if(eof == 3){
363           //eof has arrived
364           msg.eof = 1;
365           writeMessageToQueue(msg,1);
```

```
366             break;
367           }
368         }else{
369           if(eof){
370             int j;
371             for(j = 0; j < eof; j++){
372               msg.characteres[0] = '0';
373               writeMessageToQueue(msg,1);
374             }
375           }
376           if(eof == 3){
377             break;
378           }
379           msg.characteres[0] = byte[count];
380           writeMessageToQueue(msg, 1);
381           eof = 0;
382         }
383       }
384     }
385 }
386 void createThread(threadParameters * dataThread,int type){
387
388   if(type == 1){
389     pthread_create(&readsMessage, NULL,
          readMessageQueueWriteSocket , dataThread);
390   }else{
391     pthread_create(&readsMessage, NULL,
          readSocketWritesMessageQueue , dataThread);
392   }
393 }
```

A.3 Programas Principais

A.3.1 Cifrador Algoritmo Proposto

```
1 #include <functions.h>
2 #include <connections.h>
3 #include <time.h>
4 #include <sys/time.h>
5
6
7 int main(int argc, char *argv[]){
```

```
8
9   connectServer(argc, argv);
10  char key[] = {"pAatOAgJqpdy7CECQbRojYLqy11O2W38nJm+
       Sfcz94QrNAH3EXhKFxx4t0FsjfiadGDlHkdLMG8jvgjG5jlNI72gkODO9IroRk
       +
       vy151xeBQoout9ZQ6wZktuqibySxKxGy7g0RcgfW6D7vnpV9bNfsxtOZDirJOhgQWErOq
       +GKYze25B9g0gheVEumbBpgz376VwJoHPZcFRVQw6lOc/EAaozgS1Uwh1/
       Zs1ThagZC4EP85ATVPfFJ45xNUAD1LsLqGjSOc1Y7MBU7EINwXDrDJIeKeyn+
       txKu0EoJ9QAgZyXqK4gOeUyZ1br1NMZ222Cgx+
       Q5unZA6qq8zUrHsi024wldyLJL0uqHpbHn6k3IrNrNcw45BLKBbsHf4JICaPIr825XeQb0
       +VrNUS9eSCeErUIF7iNxryaW2H67EXKds2R7pGMh+
       JlogxUUMbJ5THV5HcV2rSPLugeCWDW8MACwW2Q3d6E2t1/
       v4recq3IohzX4rBoDTz4Px91wCajdgPdJpVwP82Cyv210fqI4KBZidEgtqEzJbbh7fQvGc
       +iVGhCXs5S6XihWadbfa2XyM5hF56O3kZNFG0UITbd0/
       cClVtfA7PDNEgg2pZNjVoNvwuR+/85
       b0YhDnlX6xawNBdO6xBUuRSs5oHvmd2aTTphlxlnjAlXnfgW9Oo0ACq5YbDgo
       +npCkcE4xEyO19Pf92YORUbbk4j+
       RJHcCuDqC0cdNDN5HGSTWvJn2KbzPpdg3UAcrLc7jd4usrTuiJ7QVu6uXT6FOl2h7hMGEy
       /1kol4v5KChHq6iQ5/jkQV2049c4kN03BLceIXkyXs+y5hyVlmx/
       QBNYa3etsAiw58yMVyojsB0AWgVnUkT00IY3nw2UIgaj5ENZtcZz/
       yhRNJOwDQwG4d0n2IHZYmwgD47wXW20g1o3avaThsshBMiI6dMmcogpb2R8gtCkuGvkjwl
       /8cjGAA+wcVaOU9EndKUqT56uB4hcSQ96N63n1w07D1h6AK/
       CuELXE5wNtgPzQkgi83r+6qrY/
       rFeeO0dZAUUlR01Pyw15oGw7nmRBMBR2lak1Cjsi84I9K4JyW4t/
       LEuIvc122FB7YNCg9mRbTVMnUi3vWWcvVNur6/WQBtQllSdQ9Moo/
       mfYHA0TIyf9DBWBHtaJoWP7KvVfAgvjYyMU90ijiLDrASY8580Q1g3j6LioLEaXijfGst3
       +c7K2gTk+V1KBZEEphGXQgFrJClZwoz/
       zohtn53xEooCME2cDEVaDvBf2B9Wch1BaOFJIcJdJ8JrheUM3l6eUBuUtoy5XNHb9oJsWf
       +azAOBU3mOEcXw7ZxoCNack1M/6atZOYkNUO9/zSnN/moVrYfqs+
       hcdUUJkZ4zk3tKdRVmFpohfW2dDF5+
       GhPZxlLDkogfSkQYZ5De6gNkwUi8qI3T0IaHjKs8tYOJU3+
       hlMSLSfUTwKn4JEx+V7FUiAUP0xI1Cxh5ySRQQoJqOA9eNj85"};
11
12  int messageCipher = 1;
13  int rc = createMessageQueue("cipher", messageCipher);
14
15  if(rc !=  0){
16    printf("Problem creating message queue");
17    return 1;
18  }
19
20  FILE *fp;
```

```
21  fp = fopen("teste.txt", "r");
22
23  // Starts thread cipher side
24  threadParameters dataThread;
25  dataThread.fileName = "cipher";
26  dataThread.fileOut = "cipher.txt";
27  dataThread.eof = 0;
28  int type = 1;
29  createThread(&dataThread, type);
30
31  //Message struct
32  message msg;
33  msg.type = 1;
34
35  char plaintext;
36
37  RC4 params;
38
39  int result, countTable = 0, count = 0;
40
41  resetTable();
42  rc4Initializer(&params, key);
43  signal = rc4Generator(&params);
44  table[signal] = '1';
45
46  while( ( plaintext = fgetc(fp) ) != EOF ){
47   if(countTable > 254){
48     resetTable();
49     signal = rc4Generator(&params);
50     table[signal] = '1';
51     countTable = 0;
52   }
53    do{
54      result = cipherAlgorithm(&params, plaintext);
55      if(result == 0){
56        countTable++;
57        break;
58      }else{
59        printf("ERROR");
60      }
61
62    }while(1);
```

```
63    }
64
65    msg.eof = 1;
66    writeMessageToQueue(msg, count);
67    fclose(fp);
68
69    printf("Waiting thread ends\n");
70    do{
71      usleep(100);
72    }while(dataThread.eof == 0);
73
74    destroyMessageQueue();
75
76
77  }
```

A.3.2 Decifrador Algoritmo Proposto

```
1  #include <functions.h>
2  #include <connections.h>
3
4  int main(int argc, char *argv[]){
5
6    createServer(argc, argv);
7    acceptClient();
8
9    char key[] = {"pAatOAgJqpdy7CECQbRojYLqy11O2W38nJm+
       Sfcz94QrNAH3EXhKFxx4t0FsjfiadGDlHkdLMG8jvgjG5jlNI72gkODO9IroRk
       +
       vy151xeBQoout9ZQ6wZktuqibySxKxGy7g0RcgfW6D7vnpV9bNfsxtOZDirJOhgQWErOq
       +GKYze25B9g0gheVEumbBpgz376VwJoHPZcFRVQw6lOc/EAaozgS1Uwh1/
       Zs1ThagZC4EP85ATVPfFJ45xNUAD1LsLqGjSOc1Y7MBU7EINwXDrDJIeKeyn+
       txKu0EoJ9QAgZyXqK4gOeUyZ1br1NMZ222Cgx+
       Q5unZA6qq8zUrHsi024wldyLJL0uqHpbHn6k3IrNrNcw45BLKBbsHf4JICaPIr825XeQb0
       +VrNUS9eSCeErUIF7iNxryaW2H67EXKds2R7pGMh+
       JlogxUUMbJ5THV5HcV2rSPLugeCWDW8MACwW2Q3d6E2t1/
       v4recq3IohzX4rBoDTz4Px91wCajdgPdJpVwP82Cyv210fqI4KBZidEgtqEzJbbh7fQvGc
       +iVGhCXs5S6XihWadbfa2XyM5hF56O3kZNFG0UITbd0/
       cClVtfA7PDNEgg2pZNjVoNvwuR+/85
       b0YhDnlX6xawNBdO6xBUuRSs5oHvmd2aTTphlxlnjAlXnfgW9Oo0ACq5YbDgo
       +npCkcE4xEyO19Pf92YORUbbk4j+
       RJHcCuDqC0cdNDN5HGSTWvJn2KbzPpdg3UAcrLc7jd4usrTuiJ7QVu6uXT6FOl2h7hMGEy
```

```
    /1kol4v5KChHq6iQ5/jkQV2049c4kN03BLceIXkyXs+y5hyVlmx/
    QBNYa3etsAiw58yMVyojsB0AWgVnUkT00IY3nw2UIgaj5ENZtcZz/
    yhRNJOwDQwG4d0n2IHZYmwgD47wXW20g1o3avaThsshBMiI6dMmcogpb2R8gtCkuGvkjwIkCed
    /8cjGAA+wcVaOU9EndKUqT56uB4hcSQ96N63n1w07D1h6AK/
    CuELXE5wNtgPzQkgi83r+6qrY/
    rFeeO0dZAUUlR01Pyw15oGw7nmRBMBR2lak1Cjsi84I9K4JyW4t/
    LEuIvc122FB7YNCg9mRbTVMnUi3vWWcvVNur6/WQBtQllSdQ9Moo/
    mfYHA0TIyf9DBWBHtaJoWP7KvVfAgvjYyMU90ijiLDrASY8580Q1g3j6LioLEaXijfGst3Rp
    +c7K2gTk+V1KBZEEphGXQgFrJClZwoz/
    zohtn53xEooCME2cDEVaDvBf2B9Wch1BaOFJIcJdJ8JrheUM3l6eUBuUtoy5XNHb9oJsWfZ
    +azAOBU3mOEcXw7ZxoCNack1M/6atZOYkNUO9/zSnN/moVrYfqs+
    hcdUUJkZ4zk3tKdRVmFpohfW2dDF5+
    GhPZxlLDkogfSkQYZ5De6gNkwUi8qI3T0IaHjKs8tYOJU3+
    hlMSLSfUTwKn4JEx+V7FUiAUP0xI1Cxh5ySRQQoJqOA9eNj85"};
  int messageDecipher = 2;
  int rc = createMessageQueue("decipher", messageDecipher);
  if(rc !=  0){
    printf("Problem creating message queue");
    return 1;
  }

  // Starts thread decipher side
  threadParameters dataThreadDecipher;
  dataThreadDecipher.fileName = "test";
  dataThreadDecipher.fileOut = "cipher.txt";

  message msgCipher;
  msgCipher.type = 1;
  int type = 2;
  createThread(&dataThreadDecipher, type);

  //Message struct
  message msg;
  msg.type = 1;
  char cipherText;
  RC4 paramsDecipher;

  rc4Initializer(&paramsDecipher, key);

  decipherAlgorithm(&paramsDecipher);

  destroyMessageQueue();
```

```
38 }
```

A.3.3 Cifrador *RC4*

```
1  #include <functions.h>
2  #include <time.h>
3  #include <sys/time.h>
4  #include <connections.h>
5
6
7  int main(int argc, char *argv[]){
8
9    connectServer(argc, argv);
10   char key[] = {"pAatOAgJqpdy7CECQbRojYLqy11O2W38nJm+
       Sfcz94QrNAH3EXhKFxx4t0FsjfiadGDlHkdLMG8jvgjG5jlNI72gkODO9IroRk
       +
       vy151xeBQoout9ZQ6wZktuqibySxKxGy7g0RcgfW6D7vnpV9bNfsxtOZDirJOhgQWErOq
       +GKYze25B9g0gheVEumbBpgz376VwJoHPZcFRVQw6lOc/EAaozgS1Uwh1/
       Zs1ThagZC4EP85ATVPfFJ45xNUAD1LsLqGjSOc1Y7MBU7EINwXDrDJIeKeyn+
       txKu0EoJ9QAgZyXqK4gOeUyZ1br1NMZ222Cgx+
       Q5unZA6qq8zUrHsi024wldyLJL0uqHpbHn6k3IrNrNcw45BLKBbsHf4JICaPIr825XeQb0
       +VrNUS9eSCeErUIF7iNxryaW2H67EXKds2R7pGMh+
       JlogxUUMbJ5THV5HcV2rSPLugeCWDW8MACwW2Q3d6E2t1/
       v4recq3IohzX4rBoDTz4Px91wCajdgPdJpVwP82Cyv210fqI4KBZidEgtqEzJbbh7fQvGo
       +iVGhCXs5S6XihWadbfa2XyM5hF56O3kZNFG0UITbd0/
       cClVtfA7PDNEgg2pZNjVoNvwuR+/85
       b0YhDnlX6xawNBdO6xBUuRSs5oHvmd2aTTphlxlnjAlXnfgW9Oo0ACq5YbDgo
       +npCkcE4xEyO19Pf92YORUbbk4j+
       RJHcCuDqC0cdNDN5HGSTWvJn2KbzPpdg3UAcrLc7jd4usrTuiJ7QVu6uXT6FOl2h7hMGEy
       /1kol4v5KChHq6iQ5/jkQV2049c4kN03BLceIXkyXs+y5hyVlmx/
       QBNYa3etsAiw58yMVyojsB0AWgVnUkT00IY3nw2UIgaj5ENZtcZz/
       yhRNJOwDQwG4d0n2IHZYmwgD47wXW20g1o3avaThsshBMiI6dMmcogpb2R8gtCkuGvkjwl
       /8cjGAA+wcVaOU9EndKUqT56uB4hcSQ96N63n1w07D1h6AK/
       CuELXE5wNtgPzQkgi83r+6qrY/
       rFeeO0dZAUUlR01Pyw15oGw7nmRBMBR2lak1Cjsi84I9K4JyW4t/
       LEuIvc122FB7YNCg9mRbTVMnUi3vWWcvVNur6/WQBtQllSdQ9Moo/
       mfYHA0TIyf9DBWBHtaJoWP7KvVfAgvjYyMU90ijiLDrASY8580Q1g3j6LioLEaXijfGst3
       +c7K2gTk+V1KBZEEphGXQgFrJClZwoz/
       zohtn53xEooCME2cDEVaDvBf2B9Wch1BaOFJIcJdJ8JrheUM3l6eUBuUtoy5XNHb9oJsWf
       +azAOBU3mOEcXw7ZxoCNack1M/6atZOYkNUO9/zSnN/moVrYfqs+
       hcdUUJkZ4zk3tKdRVmFpohfW2dDF5+
       GhPZxlLDkogfSkQYZ5De6gNkwUi8qI3T0IaHjKs8tYOJU3+
```

```
    hlMSLSfUTwKn4JEx+V7FUiAUP0xI1Cxh5ySRQQoJqOA9eNj85"};
  int messageCipher = 1;
  int rc = createMessageQueue("cipher", messageCipher);
  if(rc !=  0){
    printf("Problem creating message queue");
    return 1;
  }

  FILE *fp;
  fp = fopen("teste.txt", "r");

  // Starts thread cipher side
  threadParameters dataThread;
  dataThread.fileName = "cipher";
  dataThread.fileOut = "cipher.txt";
  dataThread.eof = 0;
  int type = 1;
  createThread(&dataThread, type);

  //Message struct
  message msg;
  msg.type = 1;
  char plaintext;
  RC4 params;
  int result, countTable = 0, count = 0;
  rc4Initializer(&params, key);
  while( ( plaintext = fgetc(fp) ) != EOF ){
      rc4CipherAlgorithm(&params, plaintext);
  }
  msg.eof = 1;
  writeMessageToQueue(msg, count);
  fclose(fp);

  printf("Waiting thread ends\n");
  do{
    usleep(100);
  }while(dataThread.eof == 0);

  destroyMessageQueue();

  return 0;
}
```

A.3.4 Decifrador *RC4*

```
1 #include <functions.h>
2 #include <connections.h>
3
4 int main(int argc, char *argv[]){
5
6   createServer(argc, argv);
7   acceptClient();
8
9   char key[] = {"pAatOAgJqpdy7CECQbRojYLqy11O2W38nJm+
      Sfcz94QrNAH3EXhKFxx4t0FsjfiadGDlHkdLMG8jvgjG5jlNI72gkODO9IroRk
      +
      vy151xeBQoout9ZQ6wZktuqibySxKxGy7g0RcgfW6D7vnpV9bNfsxtOZDirJOhgQWErOq
      +GKYze25B9g0gheVEumbBpgz376VwJoHPZcFRVQw6lOc/EAaozgS1Uwh1/
      Zs1ThagZC4EP85ATVPfFJ45xNUAD1LsLqGjSOc1Y7MBU7EINwXDrDJIeKeyn+
      txKu0EoJ9QAgZyXqK4gOeUyZ1br1NMZ222Cgx+
      Q5unZA6qq8zUrHsiO24wldyLJL0uqHpbHn6k3IrNrNcw45BLKBbsHf4JICaPIr825XeQb0
      +VrNUS9eSCeErUIF7iNxryaW2H67EXKds2R7pGMh+
      JlogxUUMbJ5THV5HcV2rSPLugeCWDW8MACwW2Q3d6E2t1/
      v4recq3IohzX4rBoDTz4Px91wCajdgPdJpVwP82Cyv210fqI4KBZidEgtqEzJbbh7fQvGo
      +iVGhCXs5S6XihWadbfa2XyM5hF56O3kZNFG0UITbd0/
      cClVtfA7PDNEgg2pZNjVoNvwuR+/85
      b0YhDnlX6xawNBdO6xBUuRSs5oHvmd2aTTphlxlnjAlXnfgW9Oo0ACq5YbDgo
      +npCkcE4xEyO19Pf92YORUbbk4j+
      RJHcCuDqC0cdNDN5HGSTWvJn2KbzPpdg3UAcrLc7jd4usrTuiJ7QVu6uXT6FOl2h7hMGEy
      /1kol4v5KChHq6iQ5/jkQV2049c4kN03BLceIXkyXs+y5hyVlmx/
      QBNYa3etsAiw58yMVyojsB0AWgVnUkT00IY3nw2UIgaj5ENZtcZz/
      yhRNJOwDQwG4d0n2IHZYmwgD47wXW20g1o3avaThsshBMiI6dMmcogpb2R8gtCkuGvkjwl
      /8cjGAA+wcVaOU9EndKUqT56uB4hcSQ96N63n1w07D1h6AK/
      CuELXE5wNtgPzQkgi83r+6qrY/
      rFeeO0dZAUUlR01Pyw15oGw7nmRBMBR2lak1Cjsi84I9K4JyW4t/
      LEuIvc122FB7YNCg9mRbTVMnUi3vWWcvVNur6/WQBtQllSdQ9Moo/
      mfYHA0TIyf9DBWBHtaJoWP7KvVfAgvjYyMU90ijiLDrASY8580Q1g3j6LioLEaXijfGst3
      +c7K2gTk+V1KBZEEphGXQgFrJClZwoz/
      zohtn53xEooCME2cDEVaDvBf2B9Wch1BaOFJIcJdJ8JrheUM3l6eUBuUtoy5XNHb9oJsWf
      +azAOBU3mOEcXw7ZxoCNack1M/6atZOYkNUO9/zSnN/moVrYfqs+
      hcdUUJkZ4zk3tKdRVmFpohfW2dDF5+
      GhPZxlLDkogfSkQYZ5De6gNkwUi8qI3T0IaHjKs8tYOJU3+
      hlMSLSfUTwKn4JEx+V7FUiAUP0xI1Cxh5ySRQQoJqOA9eNj85"};
10
11  int messageDecipher = 2;
```

```
12   int rc = createMessageQueue("decipher", messageDecipher);
13   if(rc !=  0){
14     printf("Problem creating message queue");
15     return 1;
16   }
17
18   // Starts thread decipher side
19   threadParameters dataThreadDecipher;
20   dataThreadDecipher.fileName = "test";
21   dataThreadDecipher.fileOut = "cipher.txt";
22   message msgCipher;
23   msgCipher.type = 1;
24
25   int type = 2;
26   createThread(&dataThreadDecipher, type);
27   int charsRead;
28
29   //Message struct
30   message msg;
31   msg.type = 1;
32   char cipherText;
33   RC4 paramsDecipher;
34
35   rc4Initializer(&paramsDecipher, key);
36
37   rc4DecipherAlgorithm(&paramsDecipher);
38
39   destroyMessageQueue();
40 }
```

APÊNDICE B – Vetor de Teste

Nesta seção, será disponibilizado exemplo de uso deste algoritmo, colocando a chave, o texto em claro e o texto cifrado correspondente.

Texto em claro:

Ashamedly aardvark towards surely a some when darn much in strategically then hey unthinking crud mindful cassowary quizzical cobra blameless oh frugal a far far rubbed the clung far eternally clever affectionately much since. Where hey llama lighted bech

Texto em claro com caracteres correspondentes da tabela ASCII:

65 115 104 97 109 101 100 108 121 32 97 97 114 100 118 97 114 107 32 116 111 119 97 114 100 115 32 115 117 114 101 108 121 32 97 32 115 111 109 101 32 119 104 101 110 32 100 97 114 110 32 109 117 99 104 32 105 110 32 115 116 114 97 116 101 103 105 99 97 108 108 121 32 116 104 101 110 32 104 101 121 32 117 110 116 104 105 110 107 105 110 103 32 99 114 117 100 32 109 105 110 100 102 117 108 32 99 97 115 115 111 119 97 114 121 32 113 117 105 122 122 105 99 97 108 32 99 111 98 114 97 32 98 108 97 109 101 108 101 115 115 32 111 104 32 102 114 117 103 97 108 32 97 32 102 97 114 32 102 97 114 32 114 117 98 98 101 100 32 116 104 101 32 99 108 117 110 103 32 102 97 114 32 101 116 101 114 110 97 108 108 121 32 99 108 101 118 101 114 32 97 102 102 101 99 116 105 111 110 97 116 101 108 121 32 109 117 99 104 32 115 105 110 99 101 46 32 87 104 101 114 101 32 104 101 121 32 108 108 97 109 97 32 108 105 103 104 116 101 100 32 98 101 99 104 10 183

Texto cifrado com caracteres correspondentes da tabela ASCII:

67 222 251 71 190 117 71 43 105 234 200 210 160 61 84 71 192 96 199 150 20 185 17 58 71 100 233 60 55 89 154 190 122 71 102 212 144 252 217 124 39 100 129 142 242 247 152 240 228 177 218 71 146 255 231 41 73 30 148 72 223 91 71 185 86 204 121 211 71 87 78 71 91 5 250 71 11 46 71 241 10 249 0 238 114 65 131 226 164 202 71 196 6 83 161 59 71 111 214 71 4 1 71 21 166 94 71 227 53 71 63 81 3 198 243 85 71 179 51 62 27 153 71 248 241 128 71 52 67 54 34 187 219 205 140 36 9 29 71 180 70 37 71 72 224 113 179 71 152 31 43 87 33 71 244 24 71 119165 71 160 183 71 188 7 236 71 218 220 71 73 116 71 222 56 16 221 71 12 76 203 74 71 250 8 71 217 237 138 2 90 71 135 146 22 193 176 75 143 71 210 69 248 71 120 126 156 18 71 4 215 125 71 20 207 88 71 240 38 03 71 159 133 106 71 237 139 71 253 169 71 121 44 118 227 64 71 222 196 13 71 99 151 71 76 208 71 160 141 135 66 216 71 64 171 229 71 57 132 93 71 44 137 71 103 245 71 64 112 71 71 201 71 48 19 174 71 176 209 194 244 50 23 120 71 68 172 253 14 71 240 181 71 26 232 71 178 173 71 150 149 71 98 163 71 255 48 35 71 31 63 71 96 127 71 106 254 71 178 246 71 92 182

71 180 107 71 26 21 71 213 195 71 148 168 71 66 180 71 43 130 71 200 155 71 191 213 71 157 230 71 85 147 71 165 49 71 184 189 145 206 71 135 98 71 93 119 15 71 82 123 71 236 239 71 109 52 71 38 28 71 56 104 71 38 175 71 27 26 71 66 184 71 121 11 71 198 12 71 73 109 71 129 45 71 118 15 8 71 213 110 71 61 157 71 189 92 71 160 159 71 215 97 77 71 26 115 71 245 4 71 82 99 71 83 170 71 88 191 71 96 235 71 234 42 40 71 194 188 71 247 136 71 125 68 71 115 47 71 118 111 71 83 192 71 255 225 71 13 101 71 170 162 71 164 79 71 141 186 71 164 25 71 57 102 71 109 17 71 252 32 71 193 134 71 37 108 71 193 95 71 115 82 71 249 71 95 57 71 26 71 130 80 71 34 197 160 160

Chave Utilizada:

WfZ+azAOBU3mOEcXw7ZxoCNack1M/6atZOYkNUO9/zSnN/moVrYf

qs+hcdUUJkZ4zk3tKdRVmFpohfW2dDF5+GhPZxlLDkogfSkQYZ5D

e6gNkwUi8qI3T0IaHjKs8tYOJU3+hlMSLSfUTwKn4JEx+V7FUiAU

P0xI1Cxh5ySRQQoJqOA9eNj85

Anexos

ANEXO A – RFC 6229

Internet Engineering Task Force (IETF)
Request for Comments: 6229
Category: Informational
ISSN: 2070-1721

J. Strombergson
SecWorks Sweden AB
S. Josefsson
Simon Josefsson Datakonsult AB

May 2011

Test Vectors for the Stream Cipher RC4

Abstract

This document contains test vectors for the stream cipher RC4.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6229.

Table of Contents

1. Introduction

The RC4 [RC4] algorithm is a widely used stream cipher. Test vectors for algorithms are useful for implementers. The RC4 cipher can use different key lengths. Advances in crypto-analysis [FMcG] [MANTIN01] [MIRONOV] [MANTIN05] suggest that initial parts of the stream output need to be discarded. This document contains several test vectors for different key lengths and for different offsets in the stream.

Motivation for this document arose from the implementation of [RFC4345].

The test vectors provided in this document have been collected by generating RC4 keystream output from three separate implementations and comparing the streams. The RC4 implementations used are Libgcrypt 1.4.4 [LIBGCRYPT], Nettle 2.0 [NETTLE], and a custom implementation.

The document contains test vectors for two different keys:

Key 1: The key byte index (starting on one), that is: 0x01, 0x02, 0x03, 0x04,...

Key 2: Generated by hashing the string "Internet Engineering Task Force" with the SHA-256 [SHS] [RFC4634] hash function, using the following command:

```
$ echo -n "Internet Engineering Task Force" | sha256sum
lada31d5cf688221c109163908ebe51debb46227c6cc8b37641910833222772a
```

The generated string has also been verified using the SHA-256 hash function implementation in OpenSSL (versions 0.9.8l and 1.0.0a) [OPENSSL].

The digest that is generated is then truncated to the appropriate length, keeping the Least Significant Bit (LSB) part of the digest as the key.

The key lengths used in this document are 40, 56, 64, 80, 128, 192, and 256 bits, respectively. The stream offsets used in this document are 0, 256, 512, 768, 1024, 1536, 2048, 3072, and 4096 bytes, respectively. Offset 1536 corresponds to recommendations in [RFC4345]. The offsets 768 and 3072 correspond to recommendations in [SANS].

2. Test Vectors for RC4

Key length: 40 bits.
key: 0x0102030405

DEC 0 HEX 0:	b2 39 63 05	f0 3d c0 27	cc c3 52 4a	0a 11 18 a8
DEC 16 HEX 10:	69 82 94 4f	18 fc 82 d5	89 c4 03 a4	7a 0d 09 19
DEC 240 HEX f0:	28 cb 11 32	c9 6c e2 86	42 1d ca ad	b8 b6 9e ae
DEC 256 HEX 100:	1c fc f6 2b	03 ed db 64	1d 77 df cf	7f 8d 8c 93
DEC 496 HEX 1f0:	42 b7 d0 cd	d9 18 a8 a3	3d d5 17 81	c8 1f 40 41
DEC 512 HEX 200:	64 59 84 44	32 a7 da 92	3c fb 3e b4	98 06 61 f6
DEC 752 HEX 2f0:	ec 10 32 7b	de 2b ee fd	18 f9 27 76	80 45 7e 22
DEC 768 HEX 300:	eb 62 63 8d	4f 0b a1 fe	9f ca 20 e0	5b f8 ff 2b
DEC 1008 HEX 3f0:	45 12 90 48	e6 a0 ed 0b	56 b4 90 33	8f 07 8d a5
DEC 1024 HEX 400:	30 ab bc c7	c2 0b 01 60	9f 23 ee 2d	5f 6b b7 df
DEC 1520 HEX 5f0:	32 94 f7 44	d8 f9 79 05	07 e7 0f 62	e5 bb ce ea
DEC 1536 HEX 600:	d8 72 9d b4	18 82 25 9b	ee 4f 82 53	25 f5 a1 30
DEC 2032 HEX 7f0:	1e b1 4a 0c	13 b3 bf 47	fa 2a 0b a9	3a d4 5b 8b
DEC 2048 HEX 800:	cc 58 2f 8b	a9 f2 65 e2	b1 be 91 12	e9 75 d2 d7
DEC 3056 HEX bf0:	f2 e3 0f 9b	d1 02 ec bf	75 aa ad e9	bc 35 c4 3c
DEC 3072 HEX c00:	ec 0e 11 c4	79 dc 32 9d	c8 da 79 68	fe 96 56 81
DEC 4080 HEX ff0:	06 83 26 a2	11 84 16 d2	1f 9d 04 b2	cd 1c a0 50
DEC 4096 HEX 1000:	ff 25 b5 89	95 99 67 07	e5 1f bd f0	8b 34 d8 75

```
Key length: 56 bits.
key: 0x01020304050607

DEC     0 HEX     0:  29 3f 02 d4  7f 37 c9 b6   33 f2 af 52  85 fe b4 6b
DEC    16 HEX    10:  e6 20 f1 39  0d 19 bd 84   e2 e0 fd 75  20 31 af c1
DEC   240 HEX    f0:  91 4f 02 53  1c 92 18 81   0d f6 0f 67  e3 38 15 4c
DEC   256 HEX   100:  d0 fd b5 83  07 3c e8 5a   b8 39 17 74  0e c0 11 d5
DEC   496 HEX   1f0:  75 f8 14 11  e8 71 cf fa   70 b9 0c 74  c5 92 e4 54
DEC   512 HEX   200:  0b b8 72 02  93 8d ad 60   9e 87 a5 a1  b0 79 e5 e4
DEC   752 HEX   2f0:  c2 91 12 46  b6 12 e7 e7   b9 03 df ed  a1 da d8 66
DEC   768 HEX   300:  32 82 8f 91  50 2b 62 91   36 8d e8 08  1d e3 6f c2
DEC  1008 HEX   3f0:  f3 b9 a7 e3  b2 97 bf 9a   d8 04 51 2f  90 63 ef f1
DEC  1024 HEX   400:  8e cb 67 a9  ba 1f 55 a5   a0 67 e2 b0  26 a3 67 6f
DEC  1520 HEX   5f0:  d2 aa 90 2b  d4 2d 0d 7c   fd 34 0c d4  58 10 52 9f
DEC  1536 HEX   600:  78 b2 72 c9  6e 42 ea b4   c6 0b d9 14  e3 9d 06 e3
DEC  2032 HEX   7f0:  f4 33 2f d3  1a 07 93 96   ee 3c ee 3f  2a 4f f0 49
DEC  2048 HEX   800:  05 45 97 81  d4 1f da 7f   30 c1 be 7e  12 46 c6 23
DEC  3056 HEX   bf0:  ad fd 38 68  b8 e5 14 85   d5 e6 10 01  7e 3d d6 09
DEC  3072 HEX   c00:  ad 26 58 1c  0c 5b e4 5f   4c ea 01 db  2f 38 05 d5
DEC  4080 HEX   ff0:  f3 17 2c ef  fc 3b 3d 99   7c 85 cc d5  af 1a 95 0c
DEC  4096 HEX  1000:  e7 4b 0b 97  31 22 7f d3   7c 0e c0 8a  47 dd d8 b8

Key length: 64 bits.
key: 0x0102030405060708

DEC     0 HEX     0:  97 ab 8a 1b  f0 af b9 61   32 f2 f6 72  58 da 15 a8
DEC    16 HEX    10:  82 63 ef db  45 c4 a1 86   84 ef 87 e6  b1 9e 5b 09
DEC   240 HEX    f0:  96 36 eb c9  84 19 26 f4   f7 d1 f3 62  bd df 6e 18
DEC   256 HEX   100:  d0 a9 90 ff  2c 05 fe f5   b9 03 73 c9  ff 4b 87 0a
DEC   496 HEX   1f0:  73 23 9f 1d  b7 f4 1d 80   b6 43 c0 c5  25 18 ec 63
DEC   512 HEX   200:  16 3b 31 99  23 a6 bd b4   52 7c 62 61  26 70 3c 0f
DEC   752 HEX   2f0:  49 d6 c8 af  0f 97 14 4a   87 df 21 d9  14 72 f9 66
DEC   768 HEX   300:  44 17 3a 10  3b 66 16 c5   d5 ad 1c ee  40 c8 63 d0
DEC  1008 HEX   3f0:  27 3c 9c 4b  27 f3 22 e4   e7 16 ef 53  a4 7d e7 a4
DEC  1024 HEX   400:  c6 d0 e7 b2  26 25 9f a9   02 34 90 b2  61 67 ad 1d
DEC  1520 HEX   5f0:  1f e8 98 67  13 f0 7c 3d   9a e1 c1 63  ff 8c f9 d3
DEC  1536 HEX   600:  83 69 e1 a9  65 61 0b e8   87 fb d0 c7  91 62 aa fb
DEC  2032 HEX   7f0:  0a 01 27 ab  b4 44 84 b9   fb ef 5a bc  ae 1b 57 9f
DEC  2048 HEX   800:  c2 cd ad c6  40 2e 8e e8   66 e1 f3 7b  db 47 e4 2c
DEC  3056 HEX   bf0:  26 b5 1e a3  7d f8 e1 d6   f7 6f c3 b6  6a 74 29 b3
DEC  3072 HEX   c00:  bc 76 83 20  5d 4f 44 3d   c1 f2 9d da  33 15 c8 7b
DEC  4080 HEX   ff0:  d5 fa 5a 34  69 d2 9a aa   f8 3d 23 58  9d b8 c8 5b
DEC  4096 HEX  1000:  3f b4 6e 2c  8f 0f 06 8e   dc e8 cd cd  7d fc 58 62
```

Key length: 80 bits.
key: 0x0102030405060708090a

DEC	0	HEX	0:	ed	e3	b0	46	43	e5	86	cc	90	7d	c2	18	51	70	99	02
DEC	16	HEX	10:	03	51	6b	a7	8f	41	3b	eb	22	3a	a5	d4	d2	df	67	11
DEC	240	HEX	f0:	3c	fd	6c	b5	8e	e0	fd	de	64	01	76	ad	00	00	04	4d
DEC	256	HEX	100:	48	53	2b	21	fb	60	79	c9	11	4c	0f	fd	9c	04	a1	ad
DEC	496	HEX	1f0:	3e	8c	ea	98	01	71	09	97	90	84	b1	ef	92	f9	9d	86
DEC	512	HEX	200:	e2	0f	b4	9b	db	33	7e	e4	8b	8d	8d	c0	f4	af	ef	fe
DEC	752	HEX	2f0:	5c	25	21	ea	cd	79	66	f1	5e	05	65	44	be	a0	d3	15
DEC	768	HEX	300:	e0	67	a7	03	19	31	a2	46	a6	c3	87	5d	2f	67	8a	cb
DEC	1008	HEX	3f0:	a6	4f	70	af	88	ae	56	b6	f8	75	81	c0	e2	3e	6b	08
DEC	1024	HEX	400:	f4	49	03	1d	e3	12	81	4e	c6	f3	19	29	1f	4a	05	16
DEC	1520	HEX	5f0:	bd	ae	85	92	4b	3c	b1	d0	a2	e3	3a	30	c6	d7	95	99
DEC	1536	HEX	600:	8a	0f	ed	db	ac	86	5a	09	bc	d1	27	fb	56	2e	d6	0a
DEC	2032	HEX	7f0:	b5	5a	0a	5b	51	a1	2a	8b	e3	48	99	c3	e0	47	51	1a
DEC	2048	HEX	800:	d9	a0	9c	ea	3c	e7	5f	e3	96	98	07	03	17	a7	13	39
DEC	3056	HEX	bf0:	55	22	25	ed	11	77	f4	45	84	ac	8c	fa	6c	4e	b5	fc
DEC	3072	HEX	c00:	7e	82	cb	ab	fc	95	38	1b	08	09	98	44	21	29	c2	f8
DEC	4080	HEX	ff0:	1f	13	5e	d1	4c	e6	0a	91	36	9d	23	22	be	f2	5e	3c
DEC	4096	HEX	1000:	08	b6	be	45	12	4a	43	e2	eb	77	95	3f	84	dc	85	53

Key length: 128 bits.
key: 0x0102030405060708090a0b0c0d0e0f10

DEC	0	HEX	0:	9a	c7	cc	9a	60	9d	1e	f7	b2	93	28	99	cd	e4	1b	97
DEC	16	HEX	10:	52	48	c4	95	90	14	12	6a	6e	8a	84	f1	1d	1a	9e	1c
DEC	240	HEX	f0:	06	59	02	e4	b6	20	f6	cc	36	c8	58	9f	66	43	2f	2b
DEC	256	HEX	100:	d3	9d	56	6b	c6	bc	e3	01	07	68	15	15	49	f3	87	3f
DEC	496	HEX	1f0:	b6	d1	e6	c4	a5	e4	77	1c	ad	79	53	8d	f2	95	fb	11
DEC	512	HEX	200:	c6	8c	1d	5c	55	9a	97	41	23	df	1d	bc	52	a4	3b	89
DEC	752	HEX	2f0:	c5	ec	f8	8d	e8	97	fd	57	fe	d3	01	70	1b	82	a2	59
DEC	768	HEX	300:	ec	cb	e1	3d	e1	fc	c9	1c	11	a0	b2	6c	0b	c8	fa	4d
DEC	1008	HEX	3f0:	e7	a7	25	74	f8	78	2a	e2	6a	ab	cf	9e	bc	d6	60	65
DEC	1024	HEX	400:	bd	f0	32	4e	60	83	dc	c6	d3	ce	dd	3c	a8	c5	3c	16
DEC	1520	HEX	5f0:	b4	01	10	c4	19	0b	56	22	a9	61	16	b0	01	7e	d2	97
DEC	1536	HEX	600:	ff	a0	b5	14	64	7e	c0	4f	63	06	b8	92	ae	66	11	81
DEC	2032	HEX	7f0:	d0	3d	1b	c0	3c	d3	3d	70	df	f9	fa	5d	71	96	3e	bd
DEC	2048	HEX	800:	8a	44	12	64	11	ea	a7	8b	d5	1e	8d	87	a8	87	9b	f5
DEC	3056	HEX	bf0:	fa	be	b7	60	28	ad	e2	d0	e4	87	22	e4	6c	46	15	a3
DEC	3072	HEX	c00:	c0	5d	88	ab	d5	03	57	f9	35	a6	3c	59	ee	53	76	23
DEC	4080	HEX	ff0:	ff	38	26	5c	16	42	c1	ab	e8	d3	c2	fe	5e	57	2b	f8
DEC	4096	HEX	1000:	a3	6a	4c	30	1a	e8	ac	13	61	0c	cb	c1	22	56	ca	cc

Key length: 192 bits.
key: 0x0102030405060708090a0b0c0d0e0f101112131415161718

DEC	0	HEX	0:	05	95	e5	7f	e5	f0	bb	3c	70	6e	da	c8	a4	b2	db	11
DEC	16	HEX	10:	df	de	31	34	4a	1a	f7	69	c7	4f	07	0a	ee	9e	23	26
DEC	240	HEX	f0:	b0	6b	9b	1e	19	5d	13	d8	f4	a7	99	5c	45	53	ac	05
DEC	256	HEX	100:	6b	d2	37	8e	c3	41	c9	a4	2f	37	ba	79	f8	8a	32	ff
DEC	496	HEX	1f0:	e7	0b	ce	1d	f7	64	5a	db	5d	2c	41	30	21	5c	35	22
DEC	512	HEX	200:	9a	57	30	c7	fc	b4	c9	af	51	ff	da	89	c7	f1	ad	22
DEC	752	HEX	2f0:	04	85	05	5f	d4	f6	f0	d9	63	ef	5a	b9	a5	47	69	82
DEC	768	HEX	300:	59	1f	c6	6b	cd	a1	0e	45	2b	03	d4	55	1f	6b	62	ac
DEC	1008	HEX	3f0:	27	53	cc	83	98	8a	fa	3e	16	88	a1	d3	b4	2c	9a	02
DEC	1024	HEX	400:	93	61	0d	52	3d	1d	3f	00	62	b3	c2	a3	bb	c7	c7	f0
DEC	1520	HEX	5f0:	96	c2	48	61	0a	ad	ed	fe	af	89	78	c0	3d	e8	20	5a
DEC	1536	HEX	600:	0e	31	7b	3d	1c	73	b9	e9	a4	68	8f	29	6d	13	3a	19
DEC	2032	HEX	7f0:	bd	f0	e6	c3	cc	a5	b5	b9	d5	33	b6	9c	56	ad	a1	20
DEC	2048	HEX	800:	88	a2	18	b6	e2	ec	e1	e6	24	6d	44	c7	59	d1	9b	10
DEC	3056	HEX	bf0:	68	66	39	7e	95	c1	40	53	4f	94	26	34	21	00	6e	40
DEC	3072	HEX	c00:	32	cb	0a	1e	95	42	c6	b3	b8	b3	98	ab	c3	b0	f1	d5
DEC	4080	HEX	ff0:	29	a0	b8	ae	d5	4a	13	23	24	c6	2e	42	3f	54	b4	c8
DEC	4096	HEX	1000:	3c	b0	f3	b5	02	0a	98	b8	2a	f9	fe	15	44	84	a1	68

Key length: 256 bits.
key: 0x0102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f20

DEC	0	HEX	0:	ea	a6	bd	25	88	0b	f9	3d	3f	5d	1e	4c	a2	61	1d	91
DEC	16	HEX	10:	cf	a4	5c	9f	7e	71	4b	54	bd	fa	80	02	7c	b1	43	80
DEC	240	HEX	f0:	11	4a	e3	44	de	d7	1b	35	f2	e6	0f	eb	ad	72	7f	d8
DEC	256	HEX	100:	02	e1	e7	05	6b	0f	62	39	00	49	64	22	94	3e	97	b6
DEC	496	HEX	1f0:	91	cb	93	c7	87	96	4e	10	d9	52	7d	99	9c	6f	93	6b
DEC	512	HEX	200:	49	b1	8b	42	f8	e8	36	7c	be	b5	ef	10	4b	a1	c7	cd
DEC	752	HEX	2f0:	87	08	4b	3b	a7	00	ba	de	95	56	10	67	27	45	b3	74
DEC	768	HEX	300:	e7	a7	b9	e9	ec	54	0d	5f	f4	3b	db	12	79	2d	1b	35
DEC	1008	HEX	3f0:	c7	99	b5	96	73	8f	6b	01	8c	76	c7	4b	17	59	bd	90
DEC	1024	HEX	400:	7f	ec	5b	fd	9f	9b	89	ce	65	48	30	90	92	d7	e9	58
DEC	1520	HEX	5f0:	40	f2	50	b2	6d	1f	09	6a	4a	fd	4c	34	0a	58	88	15
DEC	1536	HEX	600:	3e	34	13	5c	79	db	01	02	00	76	76	51	cf	26	30	73
DEC	2032	HEX	7f0:	f6	56	ab	cc	f8	8d	d8	27	02	7b	2c	e9	17	d4	64	ec
DEC	2048	HEX	800:	18	b6	25	03	bf	bc	07	7f	ba	bb	98	f2	0d	98	ab	34
DEC	3056	HEX	bf0:	8a	ed	95	ee	5b	0d	cb	fb	ef	4e	b2	1d	3a	3f	52	f9
DEC	3072	HEX	c00:	62	5a	1a	b0	0e	e3	9a	53	27	34	6b	dd	b0	1a	9c	18
DEC	4080	HEX	ff0:	a1	3a	7c	79	c7	e1	19	b5	ab	02	96	ab	28	c3	00	b9
DEC	4096	HEX	1000:	f3	e4	c0	a2	e0	2d	1d	01	f7	f0	a7	46	18	af	2b	48

Key length: 40 bits.
key: 0x833222772a

DEC	0	HEX	0:	80	ad	97	bd	c9	73	df	8a	2e	87	9e	92	a4	97	ef	da
DEC	16	HEX	10:	20	f0	60	c2	f2	e5	12	65	01	d3	d4	fe	a1	0d	5f	c0
DEC	240	HEX	f0:	fa	a1	48	e9	90	46	18	1f	ec	6b	20	85	f3	b2	0e	d9
DEC	256	HEX	100:	f0	da	f5	ba	b3	d5	96	83	98	57	84	6f	73	fb	fe	5a
DEC	496	HEX	1f0:	1c	7e	2f	c4	63	92	32	fe	29	75	84	b2	96	99	6b	c8
DEC	512	HEX	200:	3d	b9	b2	49	40	6c	c8	ed	ff	ac	55	cc	d3	22	ba	12
DEC	752	HEX	2f0:	e4	f9	f7	e0	06	61	54	bb	d1	25	b7	45	56	9b	c8	97
DEC	768	HEX	300:	75	d5	ef	26	2b	44	c4	1a	9c	f6	3a	e1	45	68	e1	b9
DEC	1008	HEX	3f0:	6d	a4	53	db	f8	1e	82	33	4a	3d	88	66	cb	50	a1	e3
DEC	1024	HEX	400:	78	28	d0	74	11	9c	ab	5c	22	b2	94	d7	a9	bf	a0	bb
DEC	1520	HEX	5f0:	ad	b8	9c	ea	9a	15	fb	e6	17	29	5b	d0	4b	8c	a0	5c
DEC	1536	HEX	600:	62	51	d8	7f	d4	aa	ae	9a	7e	4a	d5	c2	17	d3	f3	00
DEC	2032	HEX	7f0:	e7	11	9b	d6	dd	9b	22	af	e8	f8	95	85	43	28	81	e2
DEC	2048	HEX	800:	78	5b	60	fd	7e	c4	e9	fc	b6	54	5f	35	0d	66	0f	ab
DEC	3056	HEX	bf0:	af	ec	c0	37	fd	b7	b0	83	8e	b3	d7	0b	cd	26	83	82
DEC	3072	HEX	c00:	db	c1	a7	b4	9d	57	35	8c	c9	fa	6d	61	d7	3b	7c	f0
DEC	4080	HEX	ff0:	63	49	d1	26	a3	7a	fc	ba	89	79	4f	98	04	91	4f	dc
DEC	4096	HEX	1000:	bf	42	c3	01	8c	2f	7c	66	bf	de	52	49	75	76	81	15

Key length: 56 bits.
key: 0x1910833222772a

DEC	0	HEX	0:	bc	92	22	db	d3	27	4d	8f	c6	6d	14	cc	bd	a6	69	0b
DEC	16	HEX	10:	7a	e6	27	41	0c	9a	2b	e6	93	df	5b	b7	48	5a	63	e3
DEC	240	HEX	f0:	3f	09	31	aa	03	de	fb	30	0f	06	01	03	82	6f	2a	64
DEC	256	HEX	100:	be	aa	9e	c8	d5	9b	b6	81	29	f3	02	7c	96	36	11	81
DEC	496	HEX	1f0:	74	e0	4d	b4	6d	28	64	8d	7d	ee	8a	00	64	b0	6c	fe
DEC	512	HEX	200:	9b	5e	81	c6	2f	e0	23	c5	5b	e4	2f	87	bb	f9	32	b8
DEC	752	HEX	2f0:	ce	17	8f	c1	82	6e	fe	cb	c1	82	f5	79	99	a4	61	40
DEC	768	HEX	300:	8b	df	55	cd	55	06	1c	06	db	a6	be	11	de	4a	57	8a
DEC	1008	HEX	3f0:	62	6f	5f	4d	ce	65	25	01	f3	08	7d	39	c9	2c	c3	49
DEC	1024	HEX	400:	42	da	ac	6a	8f	9a	b9	a7	fd	13	7c	60	37	82	56	82
DEC	1520	HEX	5f0:	cc	03	fd	b7	91	92	a2	07	31	2f	53	f5	d4	dc	33	d9
DEC	1536	HEX	600:	f7	0f	14	12	2a	1c	98	a3	15	5d	28	b8	a0	a8	a4	1d
DEC	2032	HEX	7f0:	2a	3a	30	7a	b2	70	8a	9c	00	fe	0b	42	f9	c2	d6	a1
DEC	2048	HEX	800:	86	26	17	62	7d	22	61	ea	b0	b1	24	65	97	ca	0a	e9
DEC	3056	HEX	bf0:	55	f8	77	ce	4f	2e	1d	db	bf	8e	13	e2	cd	e0	fd	c8
DEC	3072	HEX	c00:	1b	15	56	cb	93	5f	17	33	37	70	5f	bb	5d	50	1f	c1
DEC	4080	HEX	ff0:	ec	d0	e9	66	02	be	7f	8d	50	92	81	6c	cc	f2	c2	e9
DEC	4096	HEX	1000:	02	78	81	fa	b4	99	3a	1c	26	20	24	a9	4f	ff	3f	61

```
Key length: 64 bits.
key: 0x641910833222772a

DEC     0 HEX     0:  bb f6 09 de  94 13 17 2d   07 66 0c b6  80 71 69 26
DEC    16 HEX    10:  46 10 1a 6d  ab 43 11 5d   6c 52 2b 4f  e9 36 04 a9
DEC   240 HEX    f0:  cb e1 ff f2  1c 96 f3 ee   f6 1e 8f e0  54 2c bd f0
DEC   256 HEX   100:  34 79 38 bf  fa 40 09 c5   12 cf b4 03  4b 0d d1 a7
DEC   496 HEX   1f0:  78 67 a7 86  d0 0a 71 47   90 4d 76 dd  f1 e5 20 e3
DEC   512 HEX   200:  8d 3e 9e 1c  ae fc cc b3   fb f8 d1 8f  64 12 0b 32
DEC   752 HEX   2f0:  94 23 37 f8  fd 76 f0 fa   e8 c5 2d 79  54 81 06 72
DEC   768 HEX   300:  b8 54 8c 10  f5 16 67 f6   e6 0e 18 2f  a1 9b 30 f7
DEC  1008 HEX   3f0:  02 11 c7 c6  19 0c 9e fd   12 37 c3 4c  8f 2e 06 c4
DEC  1024 HEX   400:  bd a6 4f 65  27 6d 2a ac   b8 f9 02 12  20 3a 80 8e
DEC  1520 HEX   5f0:  bd 38 20 f7  32 ff b5 3e   c1 93 e7 9d  33 e2 7c 73
DEC  1536 HEX   600:  d0 16 86 16  86 19 07 d4   82 e3 6c da  c8 cf 57 49
DEC  2032 HEX   7f0:  97 b0 f0 f2  24 b2 d2 31   71 14 80 8f  b0 3a f7 a0
DEC  2048 HEX   800:  e5 96 16 e4  69 78 79 39   a0 63 ce ea  9a f9 56 d1
DEC  3056 HEX   bf0:  c4 7e 0d c1  66 09 19 c1   11 01 20 8f  9e 69 aa 1f
DEC  3072 HEX   c00:  5a e4 f1 28  96 b8 37 9a   2a ad 89 b5  b5 53 d6 b0
DEC  4080 HEX   ff0:  6b 6b 09 8d  0c 29 3b c2   99 3d 80 bf  05 18 b6 d9
DEC  4096 HEX  1000:  81 70 cc 3c  cd 92 a6 98   62 1b 93 9d  d3 8f e7 b9

Key length: 80 bits.
key: 0x8b37641910833222772a

DEC     0 HEX     0:  ab 65 c2 6e  dd b2 87 60   0d b2 fd a1  0d 1e 60 5c
DEC    16 HEX    10:  bb 75 90 10  c2 96 58 f2   c7 2d 93 a2  d1 6d 29 30
DEC   240 HEX    f0:  b9 01 e8 03  6e d1 c3 83   cd 3c 4c 4d  d0 a6 ab 05
DEC   256 HEX   100:  3d 25 ce 49  22 92 4c 55   f0 64 94 33  53 d7 8a 6c
DEC   496 HEX   1f0:  12 c1 aa 44  bb f8 7e 75   e6 11 f6 9b  2c 38 f4 9b
DEC   512 HEX   200:  28 f2 b3 43  4b 65 c0 98   77 47 00 44  c6 ea 17 0d
DEC   752 HEX   2f0:  bd 9e f8 22  de 52 88 19   61 34 cf 8a  f7 83 93 04
DEC   768 HEX   300:  67 55 9c 23  f0 52 15 84   70 a2 96 f7  25 73 5a 32
DEC  1008 HEX   3f0:  8b ab 26 fb  c2 c1 2b 0f   13 e2 ab 18  5e ab f2 41
DEC  1024 HEX   400:  31 18 5a 6d  69 6f 0c fa   9b 42 80 8b  38 e1 32 a2
DEC  1520 HEX   5f0:  56 4d 3d ae  18 3c 52 34   c8 af 1e 51  06 1c 44 b5
DEC  1536 HEX   600:  3c 07 78 a7  b5 f7 2d 3c   23 a3 13 5c  7d 67 b9 f4
DEC  2032 HEX   7f0:  f3 43 69 89  0f cf 16 fb   51 7d ca ae  44 63 b2 dd
DEC  2048 HEX   800:  02 f3 1c 81  e8 20 07 31   b8 99 b0 28  e7 91 bf a7
DEC  3056 HEX   bf0:  72 da 64 62  83 22 8c 14   30 08 53 70  17 95 61 6f
DEC  3072 HEX   c00:  4e 0a 8c 6f  79 34 a7 88   e2 26 5e 81  d6 d0 c8 f4
DEC  4080 HEX   ff0:  43 8d d5 ea  fe a0 11 1b   6f 36 b4 b9  38 da 2a 68
DEC  4096 HEX  1000:  5f 6b fc 73  81 58 74 d9   71 00 f0 86  97 93 57 d8
```

Key length: 128 bits.
key: 0xebb46227c6cc8b37641910833222772a

DEC	0	HEX	0:	72	0c	94	b6	3e	df	44	e1	31	d9	50	ca	21	1a	5a	30
DEC	16	HEX	10:	c3	66	fd	ea	cf	9c	a8	04	36	be	7c	35	84	24	d2	0b
DEC	240	HEX	f0:	b3	39	4a	40	aa	bf	75	cb	a4	22	82	ef	25	a0	05	9f
DEC	256	HEX	100:	48	47	d8	1d	a4	94	2d	bc	24	9d	ef	c4	8c	92	2b	9f
DEC	496	HEX	1f0:	08	12	8c	46	9f	27	53	42	ad	da	20	2b	2b	58	da	95
DEC	512	HEX	200:	97	0d	ac	ef	40	ad	98	72	3b	ac	5d	69	55	b8	17	61
DEC	752	HEX	2f0:	3c	b8	99	93	b0	7b	0c	ed	93	de	13	d2	a1	10	13	ac
DEC	768	HEX	300:	ef	2d	67	6f	15	45	c2	c1	3d	c6	80	a0	2f	4a	db	fe
DEC	1008	HEX	3f0:	b6	05	95	51	4f	24	bc	9f	e5	22	a6	ca	d7	39	36	44
DEC	1024	HEX	400:	b5	15	a8	c5	01	17	54	f5	90	03	05	8b	db	81	51	4e
DEC	1520	HEX	5f0:	3c	70	04	7e	8c	bc	03	8e	3b	98	20	db	60	1d	a4	95
DEC	1536	HEX	600:	11	75	da	6e	e7	56	de	46	a5	3e	2b	07	56	60	b7	70
DEC	2032	HEX	7f0:	00	a5	42	bb	a0	21	11	cc	2c	65	b3	8e	bd	ba	58	7e
DEC	2048	HEX	800:	58	65	fd	bb	5b	48	06	41	04	e8	30	b3	80	f2	ae	de
DEC	3056	HEX	bf0:	34	b2	1a	d2	ad	44	e9	99	db	2d	7f	08	63	f0	d9	b6
DEC	3072	HEX	c00:	84	a9	21	8f	c3	6e	8a	5f	2c	cf	be	ae	53	a2	7d	25
DEC	4080	HEX	ff0:	a2	22	1a	11	b8	33	cc	b4	98	a5	95	40	f0	54	5f	4a
DEC	4096	HEX	1000:	5b	be	b4	78	7d	59	e5	37	3f	db	ea	6c	6f	75	c2	9b

Key length: 192 bits.
key: 0xc109163908ebe51debb46227c6cc8b37641910833222772a

DEC	0	HEX	0:	54	b6	4e	6b	5a	20	b5	e2	ec	84	59	3d	c7	98	9d	a7
DEC	16	HEX	10:	c1	35	ee	e2	37	a8	54	65	ff	97	dc	03	92	4f	45	ce
DEC	240	HEX	f0:	cf	cc	92	2f	b4	a1	4a	b4	5d	61	75	aa	bb	f2	d2	01
DEC	256	HEX	100:	83	7b	87	e2	a4	46	ad	0e	f7	98	ac	d0	2b	94	12	4f
DEC	496	HEX	1f0:	17	a6	db	d6	64	92	6a	06	36	b3	f4	c3	7a	4f	46	94
DEC	512	HEX	200:	4a	5f	9f	26	ae	ee	d4	d4	a2	5f	63	2d	30	52	33	d9
DEC	752	HEX	2f0:	80	a3	d0	1e	f0	0c	8e	9a	42	09	c1	7f	4e	eb	35	8c
DEC	768	HEX	300:	d1	5e	7d	5f	fa	aa	bc	02	07	bf	20	0a	11	77	93	a2
DEC	1008	HEX	3f0:	34	96	82	bf	58	8e	aa	52	d0	aa	15	60	34	6a	ea	fa
DEC	1024	HEX	400:	f5	85	4c	db	76	c8	89	e3	ad	63	35	4e	5f	72	75	e3
DEC	1520	HEX	5f0:	53	2c	7c	ec	cb	39	df	32	36	31	84	05	a4	b1	27	9c
DEC	1536	HEX	600:	ba	ef	e6	d9	ce	b6	51	84	22	60	e0	d1	e0	5e	3b	90
DEC	2032	HEX	7f0:	e8	2d	8c	6d	b5	4e	3c	63	3f	58	1c	95	2b	a0	42	07
DEC	2048	HEX	800:	4b	16	e5	0a	bd	38	1b	d7	09	00	a9	cd	9a	62	cb	23
DEC	3056	HEX	bf0:	36	82	ee	33	bd	14	8b	d9	f5	86	56	cd	8f	30	d9	fb
DEC	3072	HEX	c00:	1e	5a	0b	84	75	04	5d	9b	20	b2	62	86	24	ed	fd	9e
DEC	4080	HEX	ff0:	63	ed	d6	84	fb	82	62	82	fe	52	8f	9c	0e	92	37	bc
DEC	4096	HEX	1000:	e4	dd	2e	98	d6	96	0f	ae	0b	43	54	54	56	74	33	91

```
Key length: 256 bits.
key: 0x1ada31d5cf688221c109163908ebe51debb46227c6cc8b37641910833222772a

DEC     0 HEX     0:  dd 5b cb 00  18 e9 22 d4   94 75 9d 7c  39 5d 02 d3
DEC    16 HEX    10:  c8 44 6f 8f  77 ab f7 37   68 53 53 eb  89 a1 c9 eb
DEC   240 HEX    f0:  af 3e 30 f9  c0 95 04 59   38 15 15 75  c3 fb 90 98
DEC   256 HEX   100:  f8 cb 62 74  db 99 b8 0b   1d 20 12 a9  8e d4 8f 0e
DEC   496 HEX   1f0:  25 c3 00 5a  1c b8 5d e0   76 25 98 39  ab 71 98 ab
DEC   512 HEX   200:  9d cb c1 83  e8 cb 99 4b   72 7b 75 be  31 80 76 9c
DEC   752 HEX   2f0:  a1 d3 07 8d  fa 91 69 50   3e d9 d4 49  1d ee 4e b2
DEC   768 HEX   300:  85 14 a5 49  58 58 09 6f   59 6e 4b cd  66 b1 06 65
DEC  1008 HEX   3f0:  5f 40 d5 9e  c1 b0 3b 33   73 8e fa 60  b2 25 5d 31
DEC  1024 HEX   400:  34 77 c7 f7  64 a4 1b ac   ef f9 0b f1  4f 92 b7 cc
DEC  1520 HEX   5f0:  ac 4e 95 36  8d 99 b9 eb   78 b8 da 8f  81 ff a7 95
DEC  1536 HEX   600:  8c 3c 13 f8  c2 38 8b b7   3f 38 57 6e  65 b7 c4 46
DEC  2032 HEX   7f0:  13 c4 b9 c1  df b6 65 79   ed dd 8a 28  0b 9f 73 16
DEC  2048 HEX   800:  dd d2 78 20  55 01 26 69   8e fa ad c6  4b 64 f6 6e
DEC  3056 HEX   bf0:  f0 8f 2e 66  d2 8e d1 43   f3 a2 37 cf  9d e7 35 59
DEC  3072 HEX   c00:  9e a3 6c 52  55 31 b8 80   ba 12 43 34  f5 7b 0b 70
DEC  4080 HEX   ff0:  d5 a3 9e 3d  fc c5 02 80   ba c4 a6 b5  aa 0d ca 7d
DEC  4096 HEX  1000:  37 0b 1c 1f  e6 55 91 6d   97 fd 0d 47  ca 1d 72 b8
```

3. Security Considerations

The RC4 algorithm does not meet the basic criteria required for an encryption algorithm, as its output is distinguishable from random. The use of RC4 continues to be recommended against; in particular, its use in new specifications is discouraged. This note is intended only to aid the interoperability of existing specifications that make use of RC4.

4. Copying Conditions

This document is intended to be considered a Code Component, and is thus effectively available under the Simplified BSD License.

5. References

5.1. Normative References

[RC4] Schneier, B., "Applied Cryptography: Protocols, Algorithms, and Source Code in C", Second Edition, John Wiley and Sons, New York, NY, 1996.

5.2. Informative References

[RFC4345] Harris, B., "Improved Arcfour Modes for the Secure Shell (SSH) Transport Layer Protocol", RFC 4345, January 2006.

[RFC4634] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and HMAC-SHA)", RFC 4634, July 2006.

[FMcG] Fluhrer, S. and D. McGrew, "Statistical Analysis of the Alleged RC4 Keystream Generator", <http://www.mindspring.com/~dmcgrew/rc4-03.pdf>.

[LIBGCRYPT] Koch, W., "Libgcrypt", <http://directory.fsf.org/project/libgcrypt/>.

[MANTIN01] Mantin, I., "Analysis of the Stream Cipher RC4", <http://www.wisdom.weizmann.ac.il/~itsik/RC4/Papers/Mantin1.zip>.

[MANTIN05] Mantin, I., "Predicting and Distinguishing Attacks on RC4 Keystream Generator", Proceedings of EUROCRYPT 2005, <http://www.iacr.org/cryptodb/archive/2005/EUROCRYPT/2597/2597.pdf>.

[MIRONOV] Mironov, I., "(Not So) Random Shuffles of RC4", <http://eprint.iacr.org/2002/067.pdf>.

[NETTLE] Moeller, N., "Nettle - a low-level crypto library", <http://www.gnu.org/software/nettle/>.

[OPENSSL] OpenSSL Team, "The OpenSSL Project", <http://www.openssl.org/>.

[SANS] Hopwood, D., "Standard Cryptographic Algorithm Naming (SANS) entry on RC4", <http://www.users.zetnet.co.uk/hopwood/crypto/scan/cs.html#RC4>.

[SHS] National Institute of Standards and Technology (NIST), "FIPS Publication 180-3: Secure Hash Standard (SHS)", October 2008, <http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf>.

Authors' Addresses

Joachim Strombergson
SecWorks Sweden AB
Hogenvagen 5A
Savedalen 433 63
SE

EMail: joachim@secworks.se

Simon Josefsson
Simon Josefsson Datakonsult AB
Hagagatan 24
Stockholm 113 47
SE

EMail: simon@josefsson.org
URI: http://josefsson.org/